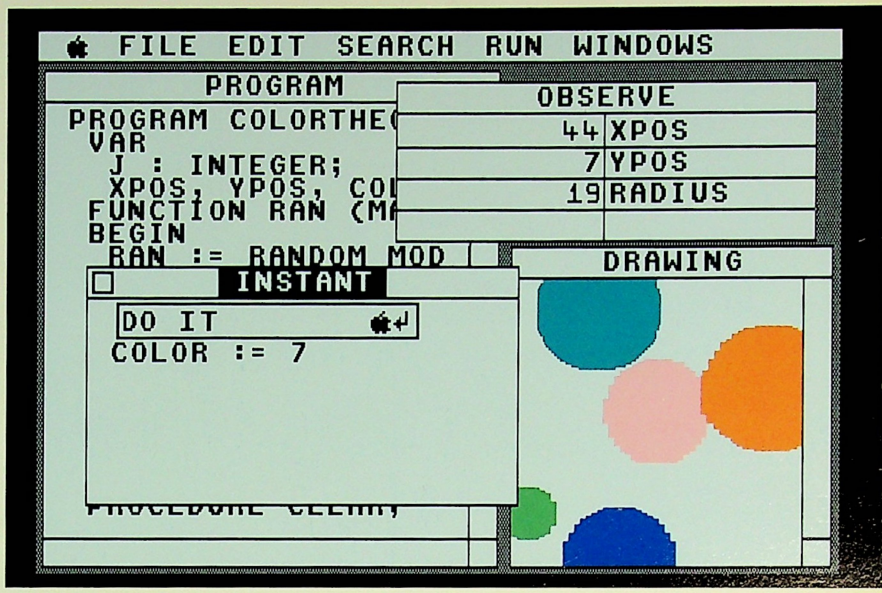
Apple® II Instant Pascal®
Language Reference Manual



Compatible with Apple IIc and 128K Apple IIe.

Copyright

Simultaneously published in the United States and Canada.

Limited Warranty on Media and Replacement

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUALS, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

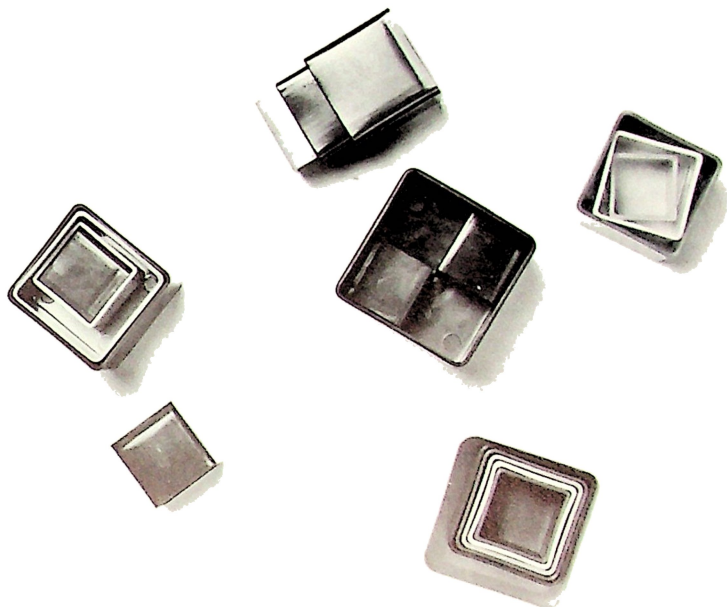
Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU THE PURCHASER ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Apple® II Instant Pascal® Language Reference Manual



Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts Menlo Park, California
Don Mills, Ontario Wokingham, England Amsterdam
Sydney Singapore Tokyo Mexico City Bogotá
Santiago San Juan

ISBN 0-201-17740-4

ABCDEFGHIJ-DO-898765
First printing, August 1985

Apple® II Instant Pascal®
Language Reference Manual

Contents

Figures and Tables

About This Manual

Welcome to the *Apple® II Instant Pascal® Language Reference Manual.* This manual supplies detailed information about the Instant Pascal version of the Pascal programming language.

This book is meant to be used along with the learning aids that come with Instant Pascal (IP):

- *Apple Presents Instant Pascal—A Training Disk.*
Apple Presents Instant Pascal is an interactive introduction to the Instant Pascal programming environment. The lessons on this disk introduce the use of IP's pull-down menus, multiple windows, and powerful debugging features.

- *The Pocket Guide to Instant Pascal.*
The *Pocket Guide to Instant Pascal* is designed for quick reference while you're programming, and as the "fast track" introduction to Instant Pascal for experienced Pascal programmers.

Who Needs This Book?

Before you begin reading this reference manual, take the time to decide whether one or all of the other IP learning aids mentioned above might better suit your needs.

For Those New to Programming

If you're a student who's using Instant Pascal as part of an introductory class, you'll find that this manual provides complete and specific information on all aspects of the language. When you begin writing your own programs, this is the place to find detailed descriptions of every feature of the language.

There are many excellent introductory books on Pascal available for novice programmers. Any one of the books listed in the Bibliography at the back of this manual would be helpful to someone who's just learning Pascal.

For Experienced Programmers

If you already have some programming experience, and you want to learn Instant Pascal without using an introductory book, you can use this manual to acquaint yourself with the language. If you're familiar with BASIC, Logo, or another version of Pascal, you can find information in this manual on how IP differs from these languages.

First, though, take the time to go through the lessons on the *Apple Presents Instant Pascal* training disk. IP is much more than a programming language—it's a complete programming environment. Even if you're an experienced user of another version of Pascal, you'll find that many of the features of the IP system are unique. The *Apple Presents Instant Pascal* disk introduces you to these powerful programming tools.

After you've completed the lessons on the training disk, take a look at the *Pocket Guide to Instant Pascal.* The *Pocket Guide* is designed to provide fast reference information on all aspects of IP. If you have extensive Pascal experience, you may be able to begin programming with IP using only the *Pocket Guide.*

What's in This Manual

Here is a brief description of the contents of this manual:

Chapter 1, "What Is Apple II Instant Pascal?" is an introduction to Instant Pascal, including an overview of how IP compares to other programming languages.

Chapter 2, "Program Components," is a brief look at the major elements of a Pascal program.

Chapters 3, "Simple Data Types," covers the use of simple data types in IP programs. This chapter contains reference information on each of the IP predefined procedures and functions that can be used with the ordinal types.

Chapter 4, "Expressions," defines the use of expressions in IP and gives detailed information on use of the arithmetic, relational, and boolean operators.

Chapter 5, "Statements," explains the use of the Pascal control structures—the statements used for branching and looping.

Chapter 6, "Procedures and Functions," covers how procedures and functions are defined within and used by IP programs. This chapter includes information on parameter passing and the rules of scope that define the accessibility of variables within a program.

Chapter 7, "Arrays, Sets, and Strings," contains information on each of these structured data types and includes information on the built-in procedures and functions that can be used with them.

Chapter 8, "Records," covers the use of this structured type, including information on the WITH statement and variant records.

Chapter 9, "Pointers and Dynamic Variables," explains some of the concepts behind dynamic variables and contains detailed information on their use in IP programs.

Chapter 10, "Introduction to Files and I/O," introduces the use of the FILE type for input and output operations from Pascal and gives reference information on each of the predefined procedures and functions used for these operations.

Appendix A, "Instant Pascal and Other Pascals," covers some of the differences between IP and Apple Pascal, Macintosh™ Pascal, and ANSI Pascal.

Appendix B, "Type Compatibility," is a detailed listing of the rules governing type compatibility.

Appendix C, "Predefined Graphics Procedures," covers the use of IP's predefined graphics procedures.

Appendix D, "Miscellaneous Predefined Procedures and Functions," describes the predefined functions and procedures that allow you to write IP programs that use the Apple II's built-in speaker to generate sound, or that use a mouse or game paddles.

Appendix E, "The Standard Apple Numeric Environment," is a description of SANE, which is the foundation for all floating-point arithmetic in IP. The appendix also includes descriptions of the procedures and functions included in the SANE library.

Appendix F, "ASCII Character Set, Reserved Words, Predefined Identifiers," contains the ASCII character set and lists of IP's reserved words and predefined identifiers.

Visual Cues

Look for these visual aids throughout the manual:

Important

Information set off like this contains detailed guidelines to follow when using a particular IP feature. You'll find information here that points out restrictions and limitations on the use of the language.

IP reserved words always appear in uppercase letters. Predefined identifiers appear in lowercase italics. User-defined identifiers, such as the names of example programs, appear in uppercase and lowercase letters. Uppercase and lowercase letters are used in this case only for readability—IP isn't "case sensitive." You can use any combination of uppercase and lowercase letters you like when typing in new identifiers.

Here's a typical use of this notation:

> Notice that the procedure PromptUser uses a WHILE statement and the *read* procedure to return the information the user types in to the main program.

Notes in the margins define new terms or point to useful information contained elsewhere in this manual or in other manuals.

Words that appear in **boldface type** are sometimes defined in marginal notes and can always be found in the glossary at the back of this manual.

Syntax Diagrams

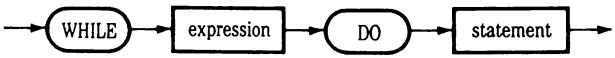
Throughout this manual, the syntax of the Instant Pascal language is illustrated with **syntax diagrams**. As with any other language, there are special rules in Instant Pascal governing the way the elements of the language are used. Syntax diagrams are an easy, shorthand way of explaining these rules.

Syntax diagram: A representation of a Pascal construct that specifies all the possible forms the construct can take.

The *Pocket Guide to Instant Pascal* contains a full set of Instant Pascal syntax diagrams.

These diagrams are easy to follow once you're used to them: begin at the upper left and follow the arrows. Every possible path through the diagram represents a valid construction in Instant Pascal. For example:

Figure P-1. Structure of a Syntax Diagram

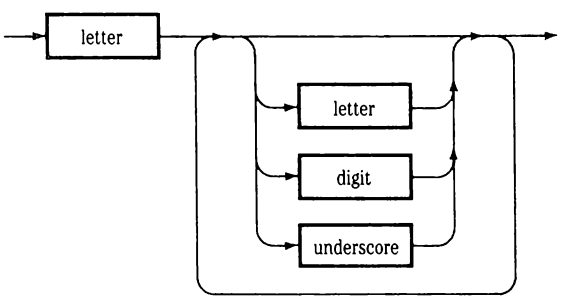


This diagram tells you that a WHILE statement consists of the word WHILE, followed by an expression, followed by the word DO, followed by a statement.

The words WHILE and DO are enclosed in rounded bubbles; this means that they are reserved words or symbols of the language, to be typed exactly as shown. The words "expression" and "statement" are in boxes with square corners; this means that they are higher-level constructions, which have their own syntax diagrams.

Here's an example where there is more than one path through the diagram:

Figure P-2. Identifier Syntax



This diagram shows that an identifier begins with a letter, and this letter may be followed by a letter, a digit, an underscore, or nothing. From here, there is the possibility of looping back to add another letter, digit, underscore, or nothing. This can be repeated indefinitely (in theory), so the syntax says that an identifier can be of any length. In practice, of course, there is a limit that the diagram can't show.

Chapter 1

What Is Apple II Instant Pascal?

Apple® II Instant Pascal® is an implementation of the Pascal language for the Apple II computer. It's based on the American National Standards Institute (ANSI) version of Pascal, which in turn is based on the original definition of Pascal by Kathleen Jensen and Niklaus Wirth in the *Pascal User Manual and Report*.

The *Pascal User Manual and Report* and many other useful books on Pascal are listed in the Bibliography at the end of this manual.

Pascal is a modern, high-level programming language that belongs to the family of structured languages. Structured languages allow you to build programs from individual blocks of code. In Pascal these blocks are called **procedures** and **functions**. Another feature of Pascal is that it gives the programmer the freedom to define unique data types.

Instant Pascal adds several important features to standard Pascal to create a powerful programming environment.

What Makes Instant Pascal Different?

Instant Pascal differs from other versions of Pascal in three major ways. The first is the way in which machine code is created and executed by the system.

An Interpreted Pascal

In the past, Pascal has always been a compiled language. When you use any language that's compiled, you must go through a two-step process before you see the results of your program. First you write the entire program, then you send it through an intermediary program called a **compiler**. The compiler takes the program text and translates it into a lower-level language (frequently machine language). The compilation process takes a fair amount of time, even for a short program. And you can't execute the program until it's been compiled.

Compiler: A language translator that converts an entire program written in a high-level programming language into an equivalent program in some lower-level language (such as machine language) for later execution.

On the other hand, some languages (like most forms of BASIC) translate source code by using an **interpreter**. An interpreter combines the two-step compile/execute process into one step. Each instruction is executed automatically as it's read by the interpreter.

Interpreter: A language translator that reads a program written in a particular programming language and immediately carries out the actions that the program describes.

One of the reasons that BASIC has been a popular language for beginning programmers is because it's interpreted rather than compiled. It's faster and easier to spot and correct errors when you use an interpreted language. You can see the results of your programming efforts without going through the two-step compilation process.

Instant Pascal is designed to give you the kind of immediate feedback you get from interpreted languages. But it does it in a slightly different way.

Each IP statement is translated into a lower-level language when it's entered into the Apple II. But it isn't executed—you do that when you select an option from the Run menu. The program still goes through the standard translate/execute sequence, but in a way that makes writing and debugging programs faster and easier.

The Instant Pascal Environment

Another major feature that sets IP apart from other languages is its unique programming environment, created by using pull-down menus and multiple windows. Instant Pascal uses these features to provide sophisticated prgramming tools that are unavailable for any other Apple II programming language.

With many languages, special debugging programs used for detecting errors in source code are available only as separate programs. IP uses the idea of multiple windows to provide built-in debugging aids, such as the Instant and Observe windows. It's faster and easier to produce error-free code using these advanced IP features.

Instant Pascal and the IEEE Floating-Point Standard

Instant Pascal offers another feature that sets it apart from most other programming languages—full conformance to IEEE Standard 754 for Floating-Point Arithmetic. The IEEE Standard is a set of guidelines used in the design and implementation of systems that perform floating-point arithmetic.

SANE is described in Appendix E.

The Standard Apple Numeric Environment (SANE) is Apple's implementation of the IEEE guidelines. Instant Pascal uses SANE to provide a powerful, flexible environment for numeric programming.

Instant Pascal and Macintosh Pascal

Instant Pascal is almost an exact subset of Macintosh™ Pascal. Most programs written in IP will run under Macintosh Pascal on a Macintosh with little or no modification.

Macintosh Pascal includes predefined procedures and functions that are not available with IP. However, almost all of the procedures and functions described in this manual are interpreted correctly by Macintosh Pascal. Exceptions are noted in the descriptions of individual procedures.

Pascal and BASIC

If you are a BASIC programmer, you will find that Pascal is different in some very fundamental ways:

- **Line Numbers:** Pascal has no line numbers. In fact, line breaks mean nothing in a Pascal program. Statements are separated from each other by semicolons. You'll find that the mechanics of writing, editing, or modifying a Pascal program are easier than with BASIC because you don't need to maintain line numbers.
- **Variables:** All variables in a Pascal program must be **declared** before they can be used. A variable declaration associates an identifier (variable name) with one of the many data types of Pascal. (In BASIC, only arrays need to be declared, via the DIM statement.)
- **Flow of Control:** Pascal has several methods for controlling the sequence in which statements are processed. These methods go beyond the IF, FOR, GOTO, and GOSUB/RETURN of BASIC. As a result, most Pascal programs are easier to read, debug, and modify than comparable BASIC programs. Pascal has a GOTO statement, but it is much less important than the other flow of control methods.
- **Procedures:** In Pascal you can write a **procedure**, which is simply a subprogram. The main program can execute the subprogram by mentioning its name. This replaces the GOSUB/RETURN mechanism of BASIC and is more powerful, since the main program can supply parameter values to the procedure when it executes it.

- **Functions:** A Pascal **function** is just a procedure that returns a value, in the same way as a BASIC user-defined function. A Pascal function definition can contain any number of statements, where a BASIC user-defined function is usually severely limited in the number of statements it can contain.
- **Block Structure:** Every Pascal program is made up of logically independent blocks of code. Each procedure or function has its own variables that are independent of the main program. A procedure or function can even have a variable of its own which has the same name as a variable in the main program. Because of this capability, a Pascal programmer can use a kind of discipline that is not possible when using most implementations of the BASIC language; the result is cleaner, more comprehensible programs.
- **Physical Addresses:** There are no POKE, PEEK, or CALL statements in Pascal. A Pascal program doesn't use physical addresses; various mechanisms in Pascal make them generally unnecessary.
- **Startup Files:** You can create BASIC startup disks using BASIC startup files. Instant Pascal programs can only be run after they have been loaded into the Program window.

Pascal and Logo

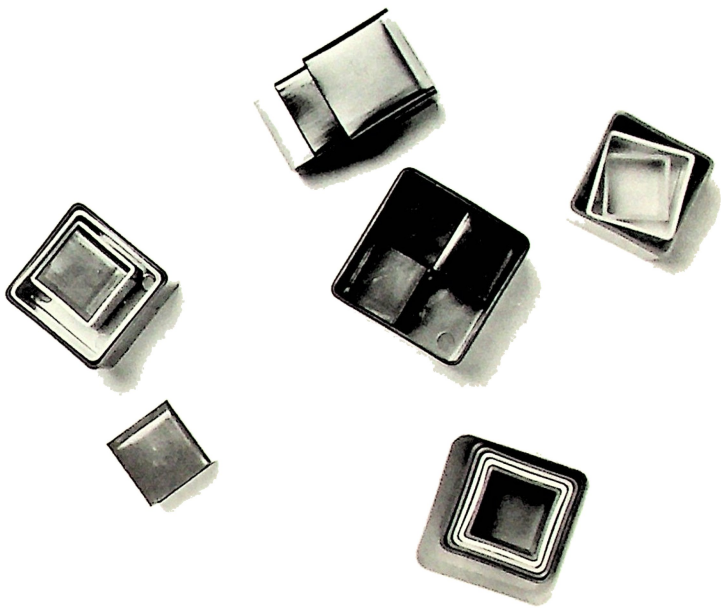
If you're familiar with Logo, these differences will be particularly noticeable:

- **Variables:** All variables in a Pascal program must be declared before they can be used. A variable declaration associates an identifier (variable name) with one of the many data types of Pascal.
- **Data Types:** Logo doesn't use data types. However, Logo programs can create and use lists, which Pascal doesn't support. Logo allows you to run these lists, a capability you won't find in Pascal.
- **Procedures:** Logo primitives and procedures are similar to Pascal's built-in procedures and functions and user-defined procedures and functions.
- **Equivalence of Procedures:** In Logo, each procedure and primitive is known to the entire program. Pascal allows nesting of blocks, which affects the scope of the identifiers declared in the block.
- **Self-Modifying Code:** Logo allows you to modify the content of a procedure as a result of running a second procedure.
- **Physical Addressing:** Apple Logo allows you to address memory directly by using special primitives.

- **Structure:** Logo programs are free form. They don't require the formal structure used by Pascal and other structured languages.
- **Startup Files:** You can create startup disks using Logo startup files. Instant Pascal programs can only be run after they have been loaded into the Program window.
- **Graphics:** Logo's turtle graphics are based on a system of relative polar coordinates. Instant Pascal provides graphics procedures and functions that use the standard Cartesian coordinate system.

Chapter 2

Program Components



This chapter describes the components that make up every Instant Pascal program, beginning with the smallest units in a program—the words, numbers, and punctuation that make up the source text of every program.

Instant Pascal Source Text

Source text: The words and characters you type into the Program Window.

Symbols: The smallest units of any programming language. Also referred to as "tokens."

The actual words you type in and see displayed in the IP Program window are called **source text**. The source text of a Pascal program is a sequence of **symbols**. Symbols are like the words, spaces, and punctuation marks that make up a paragraph of English sentences. Symbols can be divided into several types:

- Reserved words (the fixed vocabulary of Instant Pascal)
- Identifiers (names—some made up by the programmer, and others built into the language)
- Numeric constants (numbers written in the program to be used as data)
- Character and string constants (characters written in the program to be used as data)
- Delimiters (special characters and punctuation).

Reserved Words

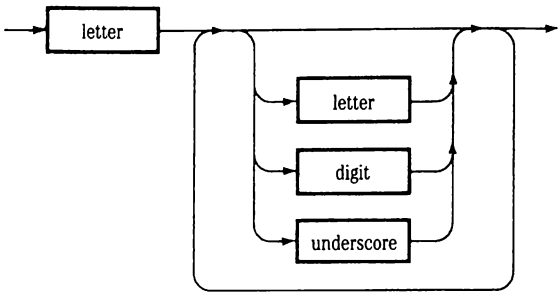
A complete list of reserved words is included in Appendix F and in the *Pocket Guide to Instant Pascal*.

Reserved words are words such as FOR, WHILE, AND, DO, and BEGIN. They make up the essential vocabulary of Pascal and have fixed meanings. Instant Pascal displays all reserved words in uppercase, boldface letters.

Identifiers

Identifiers are names for things such as variables, constants, data types, and procedures. Most identifiers are made up by the programmer and given meanings in declarations; other identifiers are names of variables, data types, procedures, and so on, that are built into the language and don't need to be declared. The syntax for an identifier is shown in Figure 2-1.

Figure 2-1. Identifier Syntax



As you can see from this diagram, an identifier must begin with a letter, and this letter may be followed by any number of letters, digits, or underscores (_). The IP interpreter ignores the case of letters; "A" and "a" are equivalent. Identifiers can be up to 255 characters in length.

Important

There is an important restriction on identifiers: an identifier must not be the same as any reserved word. Also, although it's possible to declare a new identifier that is the same as a predefined identifier, it's a good idea to avoid doing so. If you do, the new identifier overrides the predefined identifier, and the predefined identifier is no longer available for use.

See Appendix F or the *Pocket Guide* for complete lists of IP reserved words and predefined identifiers.

Numeric Constants

A **numeric constant** is any integer-type or real-type value that is written into the program to be treated as data. For example, in this statement:

```
x := 7899;
```

the number "7899" is referred to as a numeric constant. Notice that commas are not allowed in numeric constants.

A floating-point constant can be written in an Instant Pascal program as a decimal number, or in scientific notation. For example, each of the following examples is a valid representation for a floating-point constant and each represents the same value (0.637294 x 10):

```
6.37294

0.637294E1
```

```
6.37294e0

6372.94E-3

637294e-5
```

The letter "E" (or "e") and everything to its right form the exponent part of the number, which indicates multiplication by a power of ten. This is also called **floating-point notation**, because the decimal point can move, depending upon the exponent part. The syntax for an integer number is shown in Figure 2-2. The syntax for a floating-point number is shown in Figure 2-3.

Floating-point notation: A representation of a number that is comprised of an exponent part and a mantissa.

Exponent: In floating-point notation, the number that denotes the power of ten to which a number is raised.

Mantissa: In floating-point notation, the number that is multiplied by a power of ten.

Figure 2-2. Integer Number Syntax

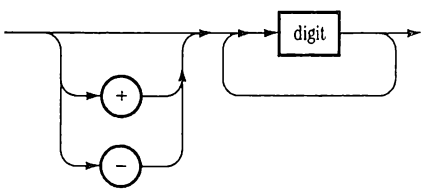
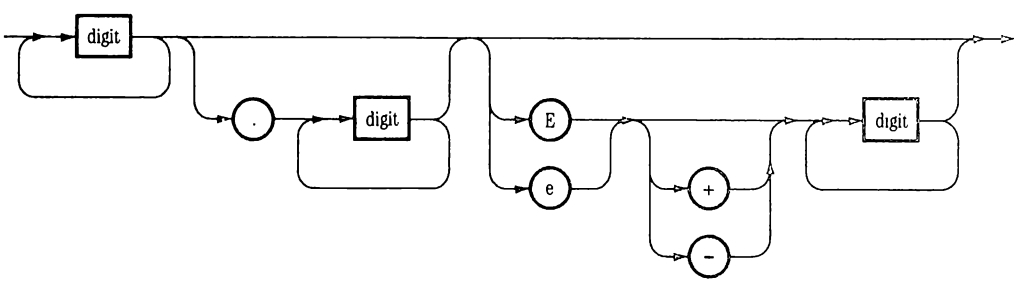


Figure 2-3. Floating-Point Number Syntax



A real-type constant may also be represented in **fixed-point notation**. Fixed-point notation is the standard decimal point representation. In the example of floating-point notation just given, the first number (6.37294) is expressed in fixed-point notation.

Character Constants

A **character constant** is a single character written into the program to be treated as data. The apostrophe, or single quotation mark ('), is used to set off character constants. The following are examples:

```
'a'
'A'
'0'
'+'
''''
```

The first two constants, 'a' and 'A', are not equivalent. In a constant, uppercase letters are distinguished from lowercase letters. The last example shows how to represent a single quotation mark as a character constant by using a series of four single quotation marks. By the way, don't confuse two single quotation marks with the double quotation mark character ("), which is not a Pascal symbol.

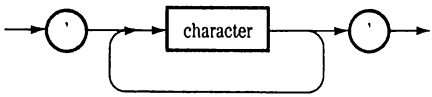
String Constants

A string constant is a character sequence written into the program to be treated as data. Like character constants, string constants are set off by apostrophes. Examples:

```
'Smith'
'$400.23'
'Type your name:  '
'PLEASE PANIC'
'DON''T PANIC'
''
```

The next-to-last example shows how to use an apostrophe as a character within a string constant. The last example shows how to represent a string consisting of no characters. The syntax for a string constant is shown in Figure 2-4.

Figure 2-4. String Constant Syntax



A string constant can also be declared with an identifier, as in the following:

```
CONST
 ErrorMessage = 'Can''t find the specified file';
```

See the section in this chapter on "Constant Definitions" for more information.

Important A string constant must be on a single line in the program; it cannot contain a line-break (carriage return).

Delimiters

Delimiter: A character that is used for punctuation to mark the beginning or end of a sequence of characters, statements, or expressions, and which therefore is not considered part of the sequence itself.

Delimiters separate symbols from each other, which is why they are called delimiters. When two symbols are not separated by a delimiter, they must be separated by a space or a line break.

In addition to serving as delimiters, many of these special characters (and a few two-character combinations) have various special meanings. They are used as the arithmetic operators, for array indexing, for setting off comments, and so on. The one-character delimiters are

, . ; : ' ( ) [ ] + − / * = < > ^ | ¦

and the two-character delimiters are

:= .. (* *) <= >= <>

The Semicolon

Pascal uses the semicolon (;) in two ways: to terminate declarations and to end statements. The semicolon is the most frequently used Pascal delimiter.

In the body of a program or procedure, Pascal uses semicolons to separate statements. When a semicolon appears at the end of a statement, it is not part of the statement: it stands alone and separates the statement from the next statement. If the next thing in the text is not a statement (for example, the word END), no semicolon is required.

Comments

A **comment** is a special instance of text that is totally ignored by the interpreter when you run your program; it serves to make the program more comprehensible not to the computer, but to a human reader.

Anything enclosed within the special symbols { and } (braces, or curly brackets) is a comment; also, anything enclosed within the special symbols (* and *) is a comment. For example:

```
{This is a comment.}
(*This is another comment.*)
```

The two types of comment delimiters can be used interchangeably. However, you cannot mix the two types of delimiters. For example:

```
(*This is not allowed.}
```

Pascal Program Structure

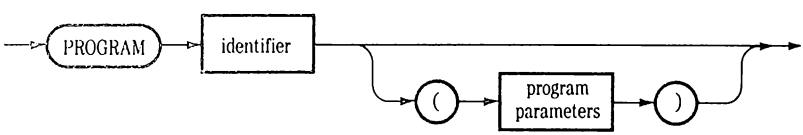
Every Pascal program has the following structure:

- The program begins with a **program heading** followed by a semicolon.
- The heading is followed by a **block**.
- The program ends with a period.

The Program Heading

A Pascal program must begin with a program heading. The heading defines the name of the program. The syntax for a program heading is shown in Figure 2-5.

Figure 2-5. Program Heading Syntax

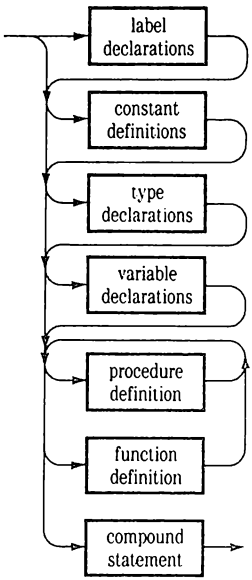


Block Structure

Pascal is a structured programming language. Unlike a free-form language, such as BASIC, Pascal allows you to define discrete modules of code whose variables are independent from the main program. These modules are called **blocks**.

Every Pascal program consists of a program heading followed by a block. The syntax of a block is shown in Figure 2-6.

Figure 2-6. Block Syntax



The USES declaration gives your program access to code that resides in a library, which is separate from the program itself. Instant Pascal comes with only one library (the SANE library, described in Appendix E). The only time you need the USES declaration is when you want to use any of the predefined procedures and functions in the SANE library.

The concept of blocks is important in Pascal. For one thing, we can now say that the outline of every program is

- program heading
- semicolon
- optional USES declaration
- block
- period

Similarly, the outline of every procedure or function definition is

- procedure or function heading
- semicolon
- block
- semicolon

Notice that every program is composed of blocks. Every block can contain procedures and functions; every procedure and function contains blocks. This means that procedures and functions can be **nested** in the same way that you nest loops.

Procedures and functions are discussed in Chapter 6.

Nested: The description of any structure that is contained within a a structure of the same kind.

The Scope of Identifiers

In a Pascal program, identifiers are known only to the block in which they were declared, and to any blocks that may be nested within that block. The **scope** of an identifier is simply the part of the program in which that identifier is known.

Remember that each piece of data being used by a block is accessed by the program through an identifier—the name of the variable or constant. The rules of scope are simple, and they apply universally to labels, declared constants, types, variables, procedures, and functions.

In this section, programs are described in terms of blocks, as defined above. You can then view any program as a structure of nested blocks. The main program itself is the outermost block, and it may have blocks nested within it; these nested blocks may have other procedures nested within them. The extent of a block is the entire block, including the heading and any blocks nested within it.

Here are the rules that govern the scope of identifiers:

- An identifier that has been used in a declaration in a particular block can be redeclared in any other block, including blocks nested within the first block.
- The scope of a declared object is the entire extent of the block in which it is declared, minus the entire extent of any nested block in which the same identifier is redeclared.
- The above rules apply to formal parameters, just as they apply to other variables declared in a procedure.

To see how this works, consider the following program fragment.

```
PROGRAM Sample;

 CONST
  Message = 'Limit exceeded.';
 VAR
  Limit:real;

 PROCEDURE TestLimit (X:  integer);
  CONST
   Limit=3;
 BEGIN
  IF X > Limit THEN
  writeln(Message)
 END;

BEGIN
  .
  .
  .
END.
```

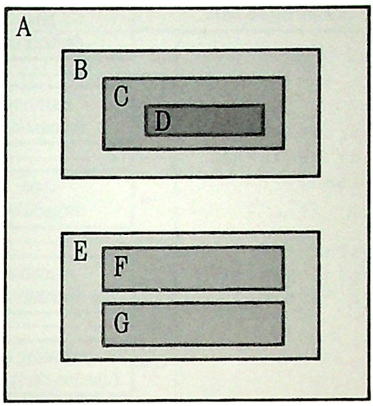
The string constant Message is known throughout the program, including the procedure TestLimit. We say that Message is **local** to the program and **global** to TestLimit.

The *integer* variable X, declared in the procedure heading of TestLimit, is known only inside TestLimit: X is local to TestLimit and unknown to the program.

The *real* variable Limit is known throughout the program, except inside TestLimit, because the integer constant Limit is declared within TestLimit. The integer constant Limit is local to TestLimit and unknown to the program—while the *real* variable Limit is local to the program and unknown to TestLimit.

The example shows a procedure within a program, but exactly the same rules apply when a procedure or function is nested within another procedure or function. Figure 2-7 illustrates the general rules of scope.

Figure 2-7. The Scope of Identifiers



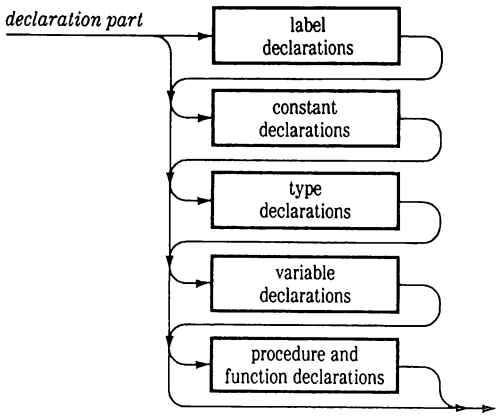
Objects Defined in Block	Are Known in Blocks
A	A, B, C, D, E, F, G
B	B, C, D
C	C, D
D	D
E	E, F, G
F	F
G	G

This scoping of identifiers becomes a real advantage when a program is large or complex. It means that you can develop a procedure without worrying about whether the procedure's variables will conflict with the main program's variables.

The Declaration Part

You can think of a Pascal program as being made up of two kinds of "sentences"—statements, which generally cause some kind of action to occur when the program is executed, and **declarations**. You use the declaration part of a block to define the nature of an identifier. The syntax for the declaration part of a Pascal program is shown in Figure 2-8.

Figure 2-8. Declaration Part Syntax



Declaration: The part of a Pascal program in which labels, constants, types, variables, procedures, and functions are defined.

The declaration part is used to declare five kinds of identifiers:

- labels
- constants
- types
- variables
- procedures and functions

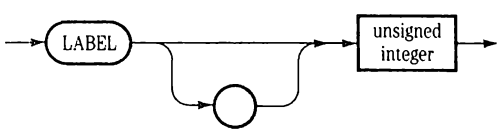
Here's a sample program that includes two kinds of declarations:

```
PROGRAM XYZ;

CONST
 MaxA = 24;                                    {An integer constant}
 MaxB = 31;                                  {Another integer constant}
 Coefficient =17.3;                             {real-type constant}

VAR
 A : integer;                                    {An integer variable}
 B : integer;                                  {Another integer variable}
 X : extended;                 {A real-type variable of type extended}

BEGIN
 FOR A := 0 TO MaxA DO
  FOR B := 0 TO MaxB DO
   BEGIN
    X := A + B * Coefficient;
    writeln(X)
   END
END.
```

Label Declarations

A label is a special kind of identifier used as an argument to the GOTO statement. A label can be any unsigned, integer number in the range 0 through 9999 (or, in Pascal notation: 0..9999). Leading zeros are not significant in labels: 0005 and 5 are equivalent. The syntax for a label declaration is shown in Figure 2-9.

Figure 2-9. Label Declaration Syntax



Constant Definitions

If you want to use a constant value in a program, you don't have to define it in the declaration part. In the example above, you could omit the constant definitions and write the body of the program including constant values, like this:

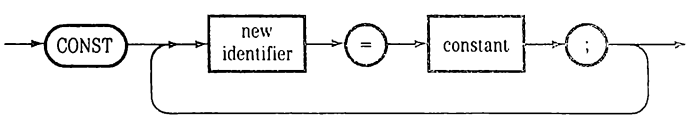
```
BEGIN
 FOR A := 0 to 24 DO
  FOR B := 0 to 31 DO
  BEGIN
   X := A + B * 17.3;
   writeln(X);
  END
END.
```

However, it's often convenient to associate a value with an identifier. If one value is used many times in a program, it's easier to change the one-line constant definition than to search the program for every occurrence of the value.

When you choose to associate a constant value with an identifier, you must define the constant in the declaration part of your program.

All constant definitions are grouped together and introduced by the reserved word CONST. The syntax for a constant definition is shown in Figure 2-10.

Figure 2-10. Constant Definition Syntax



Notice the use of the equal (=) symbol, and that each definition ends with a semicolon. In this diagram, "constant" can be any of the following:

- A signed or unsigned whole number in the range −2147483647..2147483647 representing a value of type *longint*.
- A signed or unsigned floating-point number, or an integer outside the range of type *longint* (representing a value of type *extended*).

- A character or string of characters enclosed in apostrophes, representing a value of type *char* or type *string*. A single character string constant is identical to a constant of type *char*.
- A signed or unsigned identifier of a previously defined constant (to create a new constant with the same or a negated value and a different identifier).

Chapter 7 provides more information on strings.

Type Definitions

Instant Pascal includes several predefined data types. In addition to these, you can also create additional types by defining them in the type declaration part of your program.

For example, the definition:

```
TYPE
 StringInstrument = (Bass, Cello, Viola, Violin);
```

allows you to define new variables as being of type StringInstrument. The declaration:

```
VAR
 Fiddle: StringInstrument;
```

creates a variable that can be assigned any of the values associated with type StringInstrument—and only those values. This provides automatic range checking.

New types can also be based on any of the structured types: arrays, sets, records, strings, and files. For example:

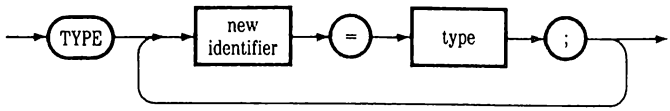
```
TYPE
 FormLetter = RECORD
 Name: STRING;
 Address: STRING;
 Zip: longint;
END;
```

This definition creates a new type called FormLetter. Using this type, you can now declare variables of this type. For example:

```
VAR
 Acceptance, Rejection, OnHold: FormLetter;
```

All type definitions are grouped together and introduced by the reserved word TYPE, as shown in Figure 2-11.

Figure 2-11. Type Definition Syntax

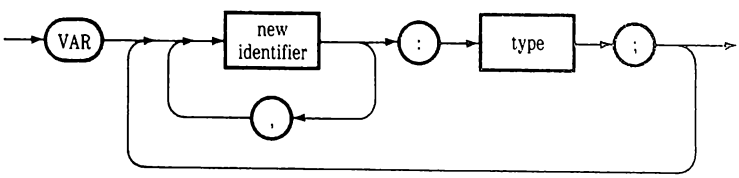


Variable Declarations

Variable: The symbol used in a program to represent a location in the computer's memory where a value can be stored.

Generally all **variables** used in a Pascal program must first be declared. (The only exceptions are dynamic variables—see Chapter 9.) When you declare a variable, you are creating an identifier, associating it with a specific data type, and allocating memory space for a value of that type. When the program is executed, the variable can take on any of a set of values depending on the type. All variable declarations are grouped together and introduced by the reserved word VAR, as shown in Figure 2-12.

Figure 2-12. Variable Declaration Syntax



The following example declares two variables:

```
VAR
 Ratio :  real;
 Iteration:  integer;
```

Notice that each declaration ends with a semicolon. The first variable, Ratio, is of type *real*, and the second, Iteration, is of type *integer*. When two or more variables of the same type are declared, you can combine the declarations:

```
VAR
 I, J, K: integer;
 X, Y, Z: real;
```

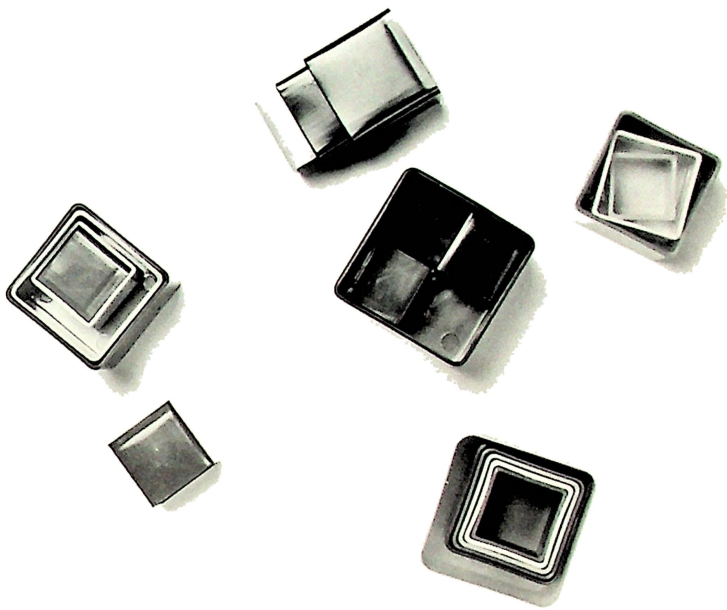
This declares three *integer* variables (I, J, and K) and three *real* variables (X, Y, and Z).

Procedure and Function Declarations

Procedure and function declarations are also included in the declaration part of a program. Procedures and functions are discussed in detail in Chapter 6.

Chapter 3

Simple Data Types



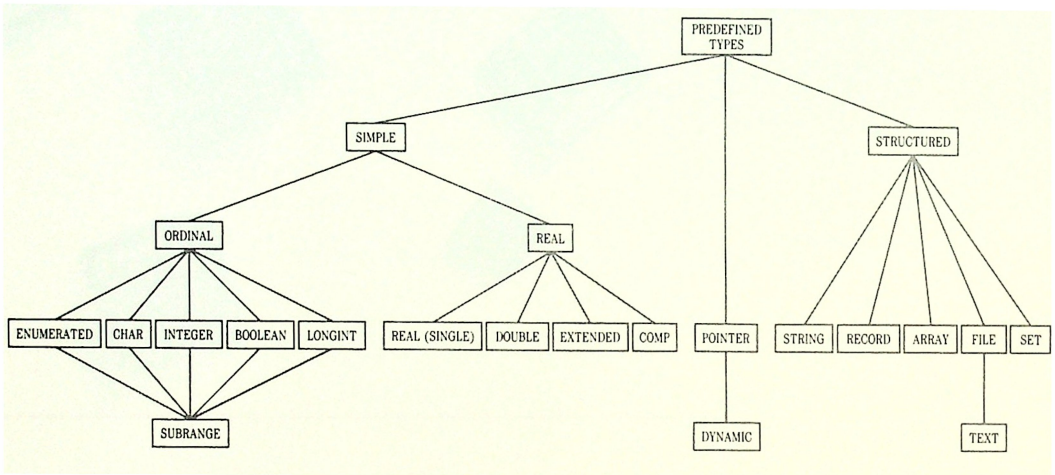
Every piece of data used or created by a Pascal program has an attribute called its **type**. The type classification tells the program and the system how to interpret each piece of data. You can think of a data type as a definition of the set of possible values that the data can have.

For example, when you use a variable of type *integer*, you can only use the variable to hold a signed, whole number value in the range −32767..32767. If you use a variable of type *boolean*, it can only hold one of the two boolean values—*true* or *false*.

Chapters 7, 8, and 10 cover the structured data types.

Instant Pascal offers a wide range of predefined types. In addition, Instant Pascal allows you to define your own types within a program. A user-defined type can be a composite of predefined types, for example, or it can be a subrange of a predefined type. This chapter describes the simple data types—the types that a single value can have.

Figure 3-1. Instant Pascal Predefined Types



The Numeric Types

Instant Pascal includes several predefined types for numeric values. These are the *integer* and *longint* types, for use with integral values, and the *real*, *double*, *extended*, and *comp* types, for use with floating-point values.

The Integer-Types

The *comp* type can be used in some cases to store integral values. See the section on the *comp* type for more information.

In Instant Pascal, integral values can be stored using one of two types: type *integer* and type *longint*. The hyphenated term "integer-type" is used to refer to either of the two types, and is hyphenated to prevent confusion with the *integer* type.

The Integer Type

The *integer* type stores whole numbers in the range −32767..32767.

Values of type *integer* can be combined with each other by means of the arithmetic operators +, −, *, DIV, and MOD to yield results of type *longint*, or by means of the / operator to yield a result of type *extended*. A value of type *integer* can also be combined arithmetically with a real-type value; when this happens, the result of the operation is type *extended*.

A value of type *integer* can be compared arithmetically with another *integer* value or with a real-type value by means of the <, <=, =, >=, >, and <> operators, to yield *boolean* results.

An *integer* value is converted to a real-type value when it is assigned to a real-type variable.

Maxint

The identifier *maxint* is the predefined identifier of a built-in constant whose value is the largest possible *integer* value, 32767. You can reference the smallest possible *integer* value, −32767, by using the negation operator (−) with *maxint*.

The Longint Type

The *longint* type is a special-purpose type. A variable of type *longint* contains a signed whole number in the range −2147483647..2147483647.

A value of type *longint* can be combined with another *longint* value by using the +, −, *, and DIV operators (but not the / and MOD operators), to produce results of type *longint*. With the same operators, a value of type *longint* can be combined with an *integer* value; the *integer* value is automatically converted to a long integer, and the result of the operation is of type *longint*. A value of type *longint* can also be combined arithmetically with a real-type value; when this happens, the result of the operation is type *extended*.

See Chapter 4 for detailed information on the arithmetic operators and how they are used with integer-type values.

When performing integer arithmetic, remember that trying to assign a *longint* value to an *integer* variable will result in an error if the *longint* value is greater than *maxint* or less than *−maxint*.

Maxlongint

The predefined constant *maxlongint* can be used to reference the largest possible *longint* value: 2147483647. You can reference the smallest possible *longint* value, −2147483647, by using the negation operator (−) with *maxlongint*.

Important

Even larger integral values can be represented by using the *comp* type. See Appendix E for more information.

The Real-Types

Instant Pascal includes four predefined real-types: *real* (or *single*), *double*, *extended*, and *comp* (or *computational*).

ANSI Pascal specifies only one type for use with floating-point values: type *real*. However, Apple has created the Standard Apple Numeric Environment (SANE) to support the standard for extended-precision arithmetic specified by the Institute of Electrical and Electronics Engineers (IEEE).

SANE includes the three types specified by the standard (*single*, *double*, and *extended*) and also includes a fourth type, *comp*, for use with accounting applications.

SANE is the basis for all floating-point values and the operations performed with them in Instant Pascal. That's why Instant Pascal has four real-types instead of only one.

SANE also includes a set of predefined functions for advanced numeric applications.

Important

SANE is described in detail in Appendix E. Instant Pascal programmers can use the *real* type, as described in this chapter, without being familiar with the level of detail in Appendix E.

Besides these types, SANE provides additional facilities for control of the floating-point environment. These facilities are included for advanced mathematical programming and aren't needed for the majority of IP programs.

However, Appendix E also contains a short overview of the philosophy behind SANE, and a description of the SANE numeric functions. Anyone using the real-types may be interested in the information presented about these topics.

Terminology

The term **real-types** refers to the set of four types used by IP for floating-point numbers. It's hyphenated to prevent confusion with the standard Pascal *real* type.

Every full ANSI Pascal must have a *real* type. For this reason, IP includes type *real*. However, it is identical to type *single*. IP accepts both terms and treats anything declared to be of either type *real* or type *single* in exactly the same way.

General Information on Real-Types

All real-type values can be combined with each other by means of the arithmetic operators +, −, *, and / to yield *extended* type results. Any real-type value can also be combined arithmetically with an integer value.

An integer value can be converted to a real-type value when it is assigned to a real-type variable. A real-type value can be converted to an integer value in one of three ways: by using the *trunc* function, the *round* function, or by using the *Num2Integer* or *Num2Longint* functions. The *round* and *trunc* functions are described later in this section; *Num2Integer* and *Num2Longint* are described in Appendix E.

A real-type value can be compared arithmetically with another real-type value or with an integer value by means of the <, <=, =, >=, >, and <> operators, to yield boolean results.

Important

Do not confuse the Instant Pascal real-types with the mathematical idea of a real number. The Pascal real-types are floating-point bit patterns or codes that can be used to represent a number in the computer. These codes and the operations on them do not always correspond exactly to their mathematical counterparts.

For example, if the exact result of an operation on real-type operands can't be represented in the 80-bit *extended* format, it is automatically rounded to fit. Subsequent calculation with this rounded value may make later results approximate as well.

See Appendix E for more information on these subjects.

Formats for the Real-Types

A general explanation of floating-point format is included in Appendix E.

The real-types are stored in binary formats, each of which includes a *sign*, an *exponent*, and a *significand*. The formats for each of the real-types are shown in Table 3-1.

Table 3-1. Floating-Point Formats

Type	Size (Bits)
Real (Single)	32
Double	64
Extended	80
Comp	64

A Note on the Comp Type

The *comp* type is a special real-type that stores integral, rather than floating-point, values. It is included in this section on the real-types because all *comp* operands are converted to *extended* before any arithmetic is performed on them, and the results of such arithmetic are always of type *extended*. An *extended* value may always be used where a *comp* value is required provided that the value, when rounded to an integral value, falls within the range of values for *comp*.

The *comp* type is intended for those applications where precise, fixed-point decimal values are required—for example, accounting applications. No explicit decimal point is ever assumed for a *comp* value, but one can be implicitly assumed by the application. For example, you can define

```
TYPE
 cents = comp;
```

and perform calculations on values of type *cents* that may also be interpreted as calculations on dollar values with an implied decimal point to the left of the second to last decimal digit.

Predefined Numeric Constants

Instant Pascal includes two predefined constants that can be used in numeric calculations: *pi* and INF.

Pi

The predefined constant *pi* represents a value of pi to the precision of the *extended* type.

INF

The constant INF represents the SANE bit pattern that is returned when a SANE operation results in an infinity, as described in Appendix E, "The Standard Apple Numeric Environment."

Numeric Functions

A set of basic numeric functions is built into Instant Pascal. Each takes a numeric value as its parameter, and returns a numeric value.

Integer-type values are automatically converted to real-type values whenever necessary. A function with a real-type formal parameter will accept an integer-type value as an actual parameter.

Important

SANE is the basis of all arithmetic performed in IP. In the following descriptions of the built-in numeric functions, remember that the term "real-type" refers to any of the SANE real-types mentioned above.

In addition to the functions described here, additional arithmetic functions are contained in the SANE library.

Full descriptions of the real-types, discussion of the rounding algorithms, and information on the additional arithmetic functions are contained in Appendix E.

The Abs Function

The *abs* function takes either a real-type value or an integer-type value as its parameter and returns a value of type *extended* (for a real-type value) or of type *longint* (for an integer-type value). The value returned is the absolute value of the parameter—that is, if the parameter is negative, the sign is changed to positive. For example,

abs(3.6545) is 3.6545
abs(−3.6545) is 3.6545
abs(−512) is 512

The Arctan Function

The *arctan* function computes the arctangent (inverse tangent) of its argument: that is, the angle whose tangent is equal to the given value. The *arctan* function takes an integer-type or real-type value as its argument and returns a value (in radians) of the *extended* type. The value returned is the principal value, in radians, of the arctangent of the argument.

The Cos Function

The *cos* function takes an integer-type or real-type value as its argument and returns a value of the *extended* type. The argument expresses a value in radians, not degrees. The value returned is the trigonometric cosine of the argument.

The Exp Function

The *exp* function takes an integer-type or real-type value as its argument and returns a value of the *extended* type. The value returned is e^x, where e is the base of the natural logarithms and x is the argument given to the function.

The Ln Function

The *ln* function takes an integer-type or real-type value as its argument and returns a value of the *extended* type. The value returned is the natural logarithm ($\log_e$) of the argument.

The Odd Function

The *odd* function takes an integer-type value as its argument and returns a *boolean* value. The result is *true* if the number is odd and *false* if the number is even.

The Random Function

The *random* function returns a uniformly distributed pseudo-random value in the range *−maxint..maxint*. The value returned depends on the global variable *randseed*, which is initialized to 14353; you can start the sequence over again from where it began by resetting *randseed* to 14353.

The Round Function

The *round* function takes a real-type value as its parameter and returns a value of type *longint*, which is obtained by rounding the real-type value to the nearest integral number. If the parameter is halfway between two integers, the result is the integral number with the greatest absolute magnitude. For example,

round(723.3) is 723
round(−5.7) is −6
round(7.7E3) is 7700
round(43.5) is 44
round(−43.5) is −44
round(42.5) is 43

An error occurs if the result is outside the range:
−maxlongint..maxlongint.

Important The *Num2Integer* and *Num2Longint* functions included in SANE provides IEEE rounding. This is described in detail in Appendix E.

The Sin Function

The *sin* function takes an integer-type or real-type value as its argument, and returns an *extended* type value. The argument expresses a value in radians, not degrees. The value returned is the trigonometric sine of the argument given.

The Sqr Function

The *sqr* function takes either a real-type value or an integer-type value as its parameter. If the argument is a real-type value, it returns an *extended* result. If the argument is an integer-type value, it returns a *longint* result. The value returned is the square of the parameter.

The Sqrt Function

The *sqrt* function takes an integer-type or real-type value as its argument, and returns an *extended* type value that is the square root of the argument. An error occurs if the argument is less than zero.

The Trunc Function

The *trunc* function takes a real-type parameter and returns a value of type *longint*. The result of the *trunc* function is the value of the argument rounded to the nearest integral number that is between zero and the argument. An error occurs if the result is outside the range of the *longint* type.

trunc(723.3) is 723
trunc(−5.7) is −5
trunc(7.7E3) is 7700

Ordinal Types

An ordinal data type has a distinct set of possible values, which are considered to be ordered in a specific way: they can be put in one-to-one correspondence with some sequence of possible values. There are four predefined ordinal types: *integer* and *longint* (described in the section on the integer-types), and *char* and *boolean*, which are described in the following sections. In addition to these predefined types, you can also create your own ordinal types.

The Char Type

A *char* value is any character from the 8-bit ASCII character set used on the Apple II. Thus the *char* values correspond to the integers from 0 to 255; the integer associated with each *char* value is called its ASCII code. The first 128 *char* values (ASCII 0 through ASCII 127) have standard interpretations as printing characters and control characters, as shown in the ASCII code table in Appendix F. The remaining char values can also be used as explained below in the section on the *char* function.

A *char* variable is declared by using the word *char* in the declaration. For example, the declaration

```
VAR
 CurrentChar, LastChar:  char;
```

declares two variables (CurrentChar and LastChar) of type *char*.

The Chr Function

In the 8-bit ASCII character set there are numerous character values that cannot be represented as *char* constants, since they cannot be entered from the keyboard. To represent such *char* values (and for other purposes), Instant Pascal provides a built-in function, *chr*. To use this function, the form is

```
chr( expression );
```

where the expression can be any expression with an integer value in the range from 0 to 255. For instance, the value of

chr(0)

is the NUL (null) character, whose ASCII code number is 0.

Important The ASCII codes in the range 128 through 255 are not assigned to specific characters, but are nevertheless usable as ASCII code values; thus *chr*(200) is a valid function call and returns the character whose ASCII code is 200 even though this does not have a standard interpretation.

The Boolean Type

The *boolean* type is an enumerated type with two values, represented by the words *false* and *true*. These words are actually identifiers for built-in constants, whose values represent a logical false result and a logical true result, respectively. These are not numerical values (though, of course, they are represented internally as binary numbers). The value *false* is less than the value *true*.

Boolean values are created in various ways; in particular, the results of comparison operations are boolean values. For example, the value of the expression

```
LegalAge > Age
```

(where LegalAge and Age are integer variables) is either *true* or *false*.

An important use for *boolean* values is in controlling the program. For example, a *boolean* value can be used to control an IF statement:

```
IF LegalAge > Age THEN
 write('Below legal age.');
```

The *write* statement is executed only if the *boolean* value of the expression LegalAge > Age is *true*.

Boolean variables are declared by using the word *boolean* in the declaration. For example, the declaration

```
VAR
  Flag1, Flag2: boolean;
```

creates two *boolean* variables, Flag1 and Flag2.

You can also declare boolean constants, by giving *false* or *true* as the value.

The *boolean* type is an ordinal type. That is, the two values are ordered—*false* and then *true*. The following relationships hold for boolean types:

false < *true*
ord(*false*) = 0
ord(*true*) = 1
succ(*false*) = *true*
pred(*true*) = *false*

Defining Enumerated Types

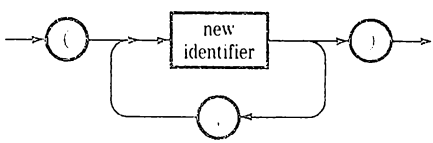
The *boolean* type is an example of a data type where the values are represented by identifiers. The meaning of these values is in the way they are used. You can define a new ordinal type by listing the identifiers for its values. This type is called an **enumerated** type. For example, the declaration

```
VAR
 Fiddle:  (Bass, Cello, Viola, Violin);
```

creates a variable, Fiddle, whose possible values are the listed identifiers from Bass to Violin. These four identifiers are constants. They correspond to the integers 0, 1, 2, 3, so that they are strictly ordered; for example, the value Cello is less than the value Viola.

The syntax for an enumerated type is shown in Figure 3-2.

Figure 3-2. Enumerated Type Syntax



Like *boolean* values, enumerated types are useful for control purposes; examples are shown in Chapter 5.

It is often useful to declare a new type explicitly. For example, instead of declaring Fiddle in the manner shown above, you could first declare the type as follows:

```
TYPE
 StringInstrument = (Bass, Cello, Viola, Violin);
```

and then declare

```
VAR
 Fiddle:  StringInstrument;
```

Subrange Types

Subrange types are used to provide automatic run-time range checking. A subrange type is based upon some ordinal type which is called the **base type**. The subrange type is exactly like the base type, except that its possible values are a subset of the possible values of the base type. For example, the declaration

```
VAR
 X:  0..255;
```

creates a variable X which is exactly like an *integer* variable except that it can only have values from 0 to 255. If the program attempts to give X a value less than 0 or greater than 255, it will be halted with an error message. Another example:

```
TYPE
 CapitalLetter = 'A' .. 'Z';
```

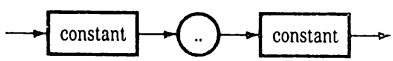
This creates the new type CapitalLetter, whose possible values are the capital letters from A through Z.

You can create a subrange type based on any ordinal type, including enumerated types. For example,

```
TYPE
 LowString = Bass .. Viola;
```

creates the type LowString, which is a subrange of the type StringInstrument in a previous example. The syntax for a subrange type is shown in Figure 3-3.

Figure 3-3. Subrange Type Syntax



Note that the two constants must be of the same type (the base type).

A subrange type is an ordinal type. The possible values of a subrange type are identical to the values of the base type that fall in the same range.

Predefined Functions for Ordinal Types

To make good use of the properties of ordinal types, Instant Pascal provides a special set of predefined functions. (Remember that a function is a subroutine that accepts one or more values as parameters, and returns one value as a result.) We have already seen the *chr* function, which takes an *integer* value as its parameter and returns the corresponding character as its result. The other special functions for ordinals are *ord*, *pred*, and *succ*.

The Ord Function

For any ordinal type, each possible value corresponds to a unique integer. This integer is called the **ordinality** of the ordinal value. For values of type *integer* or *longint*, the ordinality of each value is the same as the value itself. For all other ordinal types, the ordinalities begin at 0 for the first value and count up as far as necessary.

The *ord* function returns a *longint* value.

The *ord* function accepts a variable of any ordinal type as its parameter, and returns the ordinality of that value. The *ord* of any single character is its ASCII code. Examples:

ord(−5) is −5
ord('A') is 65 (the ASCII code for A)
ord(*false*) is 0
ord(*true*) is 1
ord(Viola) is 2 (given the declaration shown in the preceding section)

The Succ and Pred Functions

Within any ordinal type, each possible value except the last has a **successor**. The successor is simply the next value in sequence, according to the ordering of the type. Also, each possible value except the first has a **predecessor**, which is the value that precedes it. The *succ* function takes any ordinal value and returns its successor (if it has one). The *pred* function takes any ordinal value and returns its predecessor (if it has one). For example:

succ(−5) is −4
pred(c) is b
succ(Cello) is Viola (given the declaration shown above)

Important If you use the last possible value of an ordinal type as the argument for the *succ* function, or the first value of an ordinal type as the argument for the *pred* function, you will receive a runtime error.

Chapter 4

Expressions

This chapter describes the structure of Pascal expressions and each of the operators that are used to form them.

The Structure of an Expression

An expression is a combination of **operands** and **operators**. An operand is any sequence of symbols that expresses one value. Single variables can be operands, as well as lengthy combinations of variables and reserved words. The syntax for an expression and its components is shown in Figures 4-1, 4-2, 4-3, and 4-4.

Figure 4-1. Expression Syntax

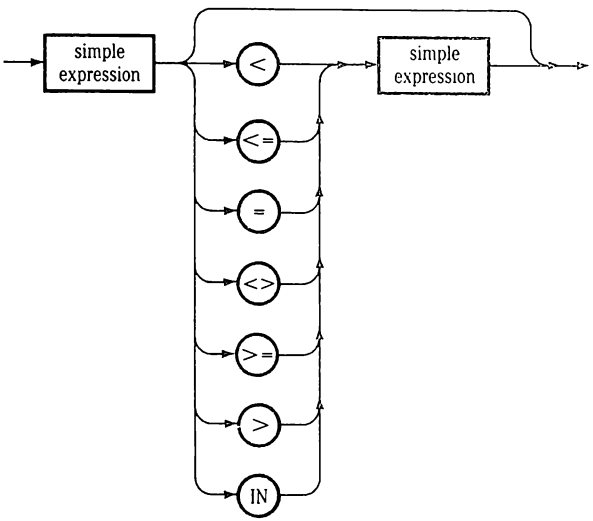


Figure 4-2. Simple Expression Syntax

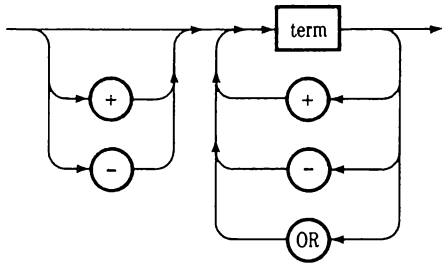


Figure 4-3. Term Syntax

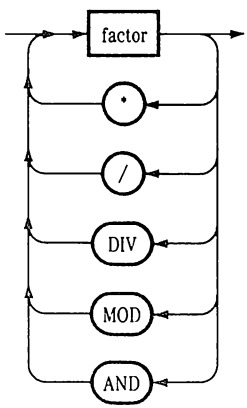
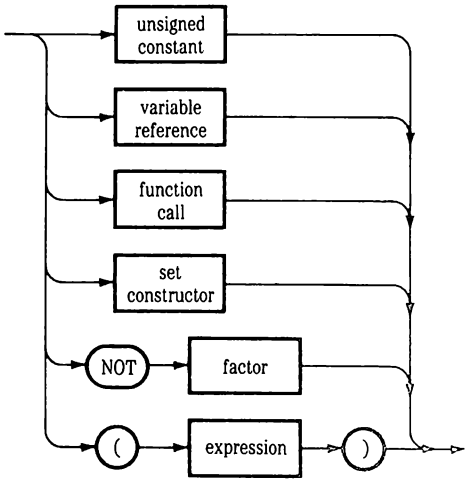


Figure 4-4. Factor Syntax



Operands and Operators

In its simplest form, an expression may be a constant, a variable reference, or a function call. More generally, an expression is a combination of **operands** and **operators**.

An operand is a single value, such as a constant, variable reference, or function call. Two operands can be combined by means of an operator, such as *, +, −, or <=. Examples:

```
X <= XLimit
RainFall.OldVal + Increment
Margin[Index] - 3
```

There are also operators that take only one operand: the logical operator NOT takes a single operand, and the + and − operators may take a single operand. These are referred to as **unary operators**. For example:

```
-X
NOT TestResult
-10
```

The simplest form of an operand is a variable reference or a constant. Here are some examples of more complex operands. Each operand on the right side of the assignment symbol is composed of an operator and two operands:

```
Result := OneOperand * AnotherOperand;
Truth  := BigInteger >= LittleInteger;
Falsity := MyDog > YourDog;
```

As these examples show, an operand itself can be an expression. Notice that this allows expressions to be complicated nestings of operations. Some slightly more sophisticated assignment statements are:

```
Pressure := N * R * Temp / Vol;
Finished := (Test = true) AND (Error = false);
Answer[K] := trunc(sqr(cos(Theta[K])));
```

Precedence of Operators

Pascal expressions often contain more than one operator. The rules of precedence determine the order in which operations within an expression will be performed. Without Pascal's rules of precedence for operators, the expression:

A − B * C / D + E

might be evaluated from left to right as if it read:

(((A − B) * C) / D) + E

But the operators do have precedence. For example, the * and / operators are always applied before the + and − operators, regardless of the sequence in the expression. The above example is actually evaluated as if it read:

(A − ((B * C) / D)) + E

The operations are grouped this way because * and / have higher precedence while + and − have lower precedence, and because operators of the same precedence are evaluated from left to right.

Pascal has four different levels of operator precedence, as shown in Table 4-1.

Table 4-1. Operator Precedence

Order of Precedence	Operators	Description
1	NOT	NOT operator
2	* / DIV MOD AND	Multiplying operators
3	+ − OR	Adding operators and signs
4	= <> < <= >= > IN	Relational operators

In this table, each line contains operators of equal precedence. The NOT operator has the highest precedence.

In an expression, all of the operators of the highest level are applied before any of the operators of the next level are applied. If an expression contains more than one operator of the same precedence, they are applied from left to right.

Important

The precedence table is one of the significant differences between Pascal and many other languages. For example, in some languages AND and OR have the same precedence. Also, in Pascal each operator symbol always has the same precedence even if it has two possible meanings; for example, the + operator can mean arithmetic addition or set union (depending on its operands), but it has the same precedence in either case. The virtue of Pascal's precedence table is that it is extremely easy to remember.

A portion of an expression can be enclosed in parentheses to form what is called a **subexpression**. A subexpression is evaluated as if it were an independent expression, before it is combined with any other parts of the expression. If there are nested parentheses (parentheses within parentheses) in the expression, the innermost subexpression is evaluated first. Parentheses can be used, as in ordinary algebra, to override the precedence of operators.

Table 4-2 gives a few examples of how expressions are evaluated:

Table 4-2. Examples of Order of Evaluation

Expression	Evaluated As	Result
4 + 3 * 2 − 1	(4 + (3 * 2)) − 1	9
8 * 2 > 5 + 6	(8 * 2) > (5 + 6)	*true*
2 > 1 AND 4 < 5	(2 > (1 AND 4)) < 5	illegal!
5 MOD 4 + 3	(5 MOD 4) + 3	4

- In the first expression the multiplying operator * is applied first, as if the expression were written 4 + (3 * 2) − 1. Because + and − are both of the same precedence, the remaining expression is evaluated from left to right, giving 9 as the result.
- The second expression is a relational operation between two arithmetic operations. The * operator is applied first, then the + operator, then the > operator. Because 8 * 2 is greater than 5 + 6, the result of the entire expression is *true*.
- The third expression appears to be a boolean operation between the results of two relational operations. The highest precedence operator, however, is AND which is applied to 1 and 4. Because AND requires boolean operands, this expression is ILLEGAL. It could be properly written as (2 > 1) AND (4 < 5).
- The fourth expression is evaluated left to right, exactly as it is written.

Important If a function call appears in an expression, the function is called, and a value is returned, before that value is used as an operand of some operator.

The Arithmetic Operators

The arithmetic operators, used with integer- and real-type operands, are shown in Table 4-3.

Table 4-3. Arithmetic Operators

Operator	Operation Performed
*	multiplication
/	division with *extended* result
DIV	division of integers with *longint* result
MOD	remainder of integer division with *longint* result
+	addition or identity
−	subtraction or negation

Each operator has specific rules concerning what type of operands it may take, and what type of results are produced. Remember that each of the numeric types has size constraints.

Important

Integer-type arithmetic overflow occurs in two instances:

- When an arithmetic operation performed on integer-type operands results in a final or intermediate result outside the limits of type *longint*.
- When a result outside the range of type *integer* is assigned to a variable of type *integer*.

Integer arithmetic overflow causes a runtime error.

Operations performed on real-type operands return results in the *extended* type. If the variable to which the result is assigned is one of the other real-types, the result is rounded, using the rounding algorithm described in Appendix E. Real-type arithmetic overflow occurs in these instances:

- When an arithmetic operation on real-type operands results in a value greater than the limits of the *extended* type.
- When a value is greater than the limits of the real-type to which it is assigned, as described in Appendix E.

However, the IEEE Standard specifies that you must be able to use standard arithmetic operations with real-types without error halts. Predeclared procedures included in the SANE library provide the ability to disable error halts. This facility is also described in Appendix E.

The Asterisk Operator

Multiplication is done with the asterisk (*) operator. Multiplication of values of either of the integer-types produces *longint* results. Multiplication involving at least one operand of any of the real-types is done with the *extended* type and the result is *extended.*

Two examples of multiplication in expressions are:

```
writeln('two cubed = ', 2*2*2);
A := C * D
```

The Slash Operator

The slash (/) operator is used for division of both real-type and integer-type operands. It returns a result of type *extended.*

Example:

```
ApproxPi := 355 / 113;
```

ApproxPi must be a real-type variable.

The DIV Operator

The DIV operator is the integer division operator. It can be used only with *integer* or *longint* operands. The result is always of type *longint.*

The *div* operation produces a result that is truncated toward 0: the remainder of the division is lost. The expression

```
A DIV B
```

(where A and B are *integer* values and B is not zero) is equivalent to

```
trunc(A / B)
```

This shows the relationship between DIV and /.

The MOD Operator

The MOD operator takes two integer-type operands and returns a value of type *longint* that is the remainder of the first operand divided by the second operand. The second operand must be positive.

A typical application of the MOD operator is

A MOD B = 0

which is *true* if B is a factor of A. Other examples:

4 MOD 3 = 1
(−4) MOD 3 = 2
0 MOD 3 = 0

Notice that since the MOD operator has a higher precedence than the unary minus operator, parentheses are needed around the "−4" to ensure that MOD operates on the value −4 rather than on the result of 4 MOD 3.

The Plus Operator

The plus (+) operator is used for addition of integer-types and real-types. The addition of two integer-type operands returns a result of type *longint*. If either or both of the operators is a real-type, the result is *extended*.

Example:

```
XReal := AInteg + XReal + 2;
```

The + operator can be used with a single operand as a sign indicator. This use of + produces a result that is identical to its operand.

The Minus Operator

The minus (−) operator is used for subtractions of integer-type and real-type operands. The subtraction of two integer operands returns a result of type *longint*. Subtraction of any of the real-types returns an *extended* result.

The − operator can also be used with a single operand (integer-type, or a real-type) to perform arithmetic negation, that is, to change the sign of the operand. For example:

−3
−A
−*trunc*(A)
−(3 + 2)
−(−2)

Important If a negated operand is used before or after one of the arithmetic multiplying operators (*, /, DIV, or MOD) the operand must be enclosed in parentheses. Two examples are:

```
(-A) * (-B)
4 * (-Offset) MOD (-(3 + 2))
```

The Relational Operators

Another relational operator, IN, is used with set values. It is explained in Chapter 7.

The relational operators are used to make comparisons between operands. They are especially useful with the flow of control statements which are explained in the next chapter. The relational operators are shown in Table 4-4.

Table 4-4. Relational Operators

>	greater than
>=	greater than or equal to
=	equal to
<	less than
<=	less than or equal to
<>	not equal to

Any real-type value can be compared with another real-type value or with an integer-type value. Otherwise a comparison that mixes types will cause an error to occur.

Boolean and user-defined ordinals may only be compared with values of the same type. Of course expressions may also be compared, provided that the comparison follows the rules described above. Here are some examples of properly used relational operators:

```
IF Scores[I] > MaxScore THEN
 MaxScore := Scores[I];

WHILE Angle < Circumference / ArcLength * 360 DO...;

REPEAT
   .
   .
   .
UNTIL Index > Limit;
```

Notice that relational operators, because they give *boolean* results, are a primary tool used to control program flow. The use of *boolean* values in IF, WHILE and REPEAT statements is explained in the next chapter.

User-defined enumerated values can be compared using any of the relational operators. The ordinality of a value of an enumerated type is determined by its position in the type's declaration. Thus, with the declaration

```
VAR
 Paint: (None, Red, Orange, Yellow);
```

the statement

```
IF Paint > None THEN
 PaintPicture;
```

will cause the procedure PaintPicture to be executed if the value of Paint is Red, Orange, or Yellow.

The result of comparing two numeric values will be exactly one of four possible relations:

- equal
- less than
- greater than
- unordered

We are used to thinking that if two numbers are unequal, then one must be larger than the other. But the real-types include, in addition to numeric values, special diagnostic values that result from invalid operations (like 0/0). These diagnostic values compare unordered with numeric values, and such comparisons may even cause runtime error. See Appendix E for details; for ordinary programming just remember that, for real-type comparisons,

- "a <> b" (that is, a is not equal to b) means a is less than, greater than, or unordered with b, and
- if "a < b" is *false*, you don't automatically know that "a >= b", because a and b may be unordered with respect to one another.

Some relational operators can also be used to compare structured types. The operations permitted between operands of the structured types (arrays, sets, strings, records, and pointers) are explained in the chapters devoted to each type.

Logical Operators

Sometimes you need to find a result that is dependent on more than one boolean value. In Pascal, this capability is provided by the logical operators, which, in order of precedence, are shown in Table 4-5.

Table 4-5. Logical Operators

NOT	boolean negation
AND	boolean conjunction
OR	boolean disjunction

Logical operators take boolean operands and produce boolean results according to the rules shown in Table 4-6.

Table 4-6. Logical Operation Results

A	B	A AND B	A OR B	NOT B
true	*true*	*true*	*true*	*false*
true	*false*	*false*	*true*	*true*
false	*true*	*false*	*true*	*false*
false	*false*	*false*	*false*	*true*.

As you can see the operator NOT takes a single boolean operand whereas the operators AND and OR both require two operands.

Examples of correctly used logical operators are:

```
(A > B) AND (C < sqr(D))
(A AND B) AND (NOT(A AND B))
(Count <= 100) OR Error
```

In the first example, `sqr(D)` returns the square of D. In the second example, the result is always *false*. In the third example, *Error* is a *boolean* variable.

Important

Although it is not always necessary to evaluate both operands of a boolean expression in order to determine the result, IP will nevertheless always evaluate both operands.

Relational Operators With Boolean Operands

Each of the relational operators (=, <>, <=, <, >, >=, IN) yields a *boolean value*. Furthermore, because *boolean* is an enumerated type, *false* < *true*. Therefore, it is possible to define each of the boolean operations using logical and relational operators. If p and q are boolean values, one can express:

implication	as p<=q
equivalence	as p=q
exclusive OR	as p<>q

Without the use of relational operators you would have to express the exclusive OR function as

```
(p AND NOT q) OR (NOT p AND q)
```

Result Types

The result types for all combinations of operator and operands are described in full detail in the preceding sections. This section summarizes those descriptions.

In Table 4-7, the column on the right lists the operators that may not be used with the pair of operands in that row.

Table 4-7. Arithmetic Operation Result Types

Operand Types	Result Type	Illegal Operators
integer-type, integer-type	*longint* (*extended* for / operator)	
real-type, real-type	*extended*	DIV, MOD
integer-type, real-type	*extended*	DIV, MOD

The logical operators NOT, AND, and OR all take *boolean* operands and give *boolean* results. The NOT operator precedes its single operand; AND and OR each take two operands.

The relational operators >, >=, =, <=, < and <> can be used to compare two *boolean* values, two enumerated values of the same type, or two numeric values. All relational operations yield *boolean* results.

Table 4-8. Simple Type Compatibility

Variable Type		**Expression Type**
integer	:=	*integer-type*
longint	:=	*integer-type*
real-type	:=	integer-type or real-type
boolean	:=	*boolean*
char	:=	*char*

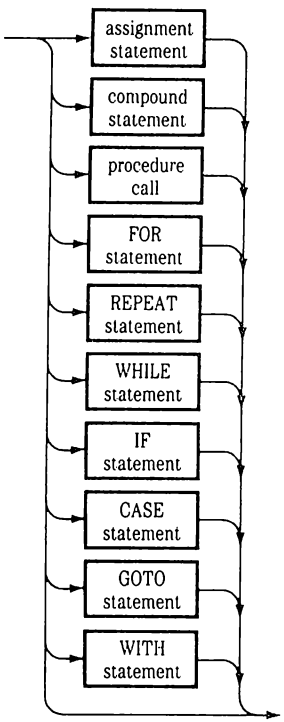
Important Appendix B lists the rules that define type compatibility. Expressions containing user-defined types are evaluated according to these rules.

Chapter 5

Statements

The major portion of a block consists of **statements**. Pascal statements do the work of a Pascal program. The syntax for a statement is shown in Figure 5-1.

Figure 5-1. Statement Syntax



Important

Technically, there is an eleventh type of Pascal statement called a null statement. This is a statement that doesn't contain anything. It simply means that whenever Pascal syntax calls for a statement, you can omit it.

It also means that when a program contains an unnecessary semicolon, the Instant Pascal interpreter considers the semicolon to be separating a null statement from another statement. The result is two statements where you intend to have only one. Most of the time this is harmless, but occasionally it causes an error because only one statement is allowed.

Except for the WITH and assignment statements, all the statements shown in the diagram are statements that determine the flow of control in a Pascal program; that is, they determine the order in which other statements are executed. There are flow of control statements that repetitively or conditionally execute other statements, and there are statements that transfer control to another portion of the program.

The flow of control statements can be subdivided into groups.

- The compound statement groups several statements into one.
- The procedure call statement causes execution of a procedure.
- The repetition statements (FOR, REPEAT, and WHILE) allow a sequence of statements to be executed repeatedly.
- The conditional statements (IF and CASE) permit conditional execution of statements.
- The GOTO statement permits unconditional transfer of control from one part of the program to another.

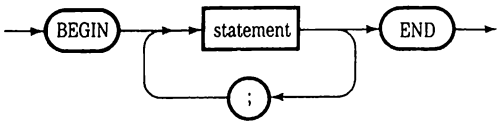
There are rules governing the scope of GOTO statements. These are discussed later in this chapter.

The Compound Statement

Every block in a Pascal program consists of a declaration part followed by a **compound statement**. In some ways this is the most important kind of statement in Pascal. It consists of the reserved word BEGIN, followed by any number of Pascal statements, followed by the reserved word END. This allows you to put any number of statements in a place where only one statement is allowed, since the compound statement is just one statement.

The syntax for a compound statement is shown in Figure 5-2.

Figure 5-2. Compound Statement Syntax



A compound statement can contain any number of statements of any type, separated by semicolons. Note that the word BEGIN does not have a semicolon after it, because it is not a statement; likewise, there is no semicolon just before the END because END is not a statement.

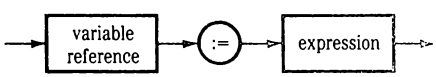
The compound statement is always considered as a single statement, even though it may contain more than one statement. Keep this in mind because most of the other flow of control statements act on single statements. Whenever you want a sequence of statements to be treated as a single statement, simply surround it with BEGIN and END.

The Assignment Statement

Assignment: The process of giving a value to a variable or function result.

One of the most commonly used symbols in Pascal expressions is the **assignment** symbol, which is used in the assignment statement. The assignment statement gives the value of an expression or a function result to a variable. The syntax of the assignment statement is shown in Figure 5-3.

Figure 5-3. Assignment Statement Syntax



The symbol := is the assignment symbol. It means that the expression on the right-hand side of the assignment operator is to be evaluated, and the result is to become the new value of the variable that is referred to on the left-hand side.

There are restrictions concerning what type of values may be assigned to what type of variables. The legal assignments for non-structured variables are shown in Table 5-1.

Table 5-1. Simple Type Compatibility

Variable Type		Expression Type
integer	:=	*integer* or *longint*
longint	:=	*integer* or *longint*
real-type	:=	integer-type or real-type
boolean	:=	*boolean*
char	:=	*char*

Important

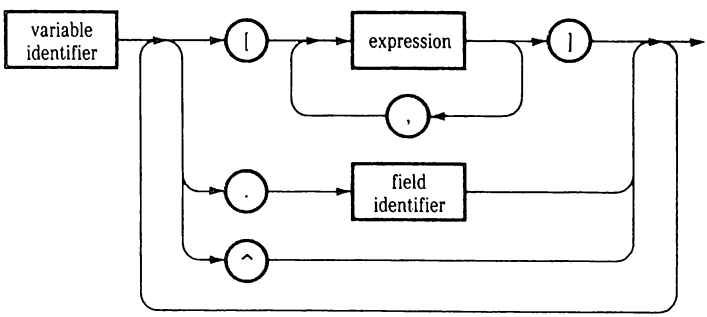
Appendix A lists the rules that define type compatibility. Expressions that contain user-defined types are evaluated according to these rules.

Variable Reference

Chapters 7 and 8 define the IP structured types referred to in some of these examples. Chapter 9 describes pointer variables. Chapter 10 explains files, which use pointer-type variables for I/O processes.

The assignment statement can also be used to assign a value to a variable of one of the structured or pointer types. In this case, the variable must reference the particular element to which the value is to be given, using the syntax shown in Figure 5-4.

Figure 5-4. Variable Reference Syntax



Here are some examples of variable reference:

X
Stadium[4]
TaxForm.Year
NewFile^

In other words, a variable reference is an identifier, which may or may not have a number of "qualifications" appended to it. Each qualification is either:

- An array subscript notation in square brackets. Stadium[4] is an array reference to the fourth element in array Stadium.
- A record field identifier set off by a period. TaxForm.Year is a reference to the Year field in the record TaxForm.
- Or the ^ (caret) symbol to indicate a variable of a pointer type. NewFile^ is a reference that points to an element of type NewFile.

The Procedure Call Statement

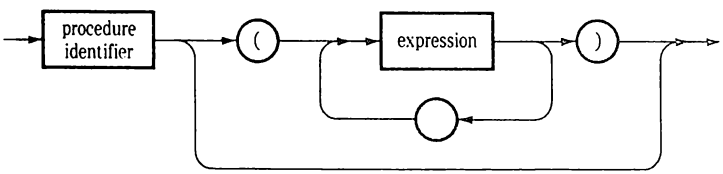
Procedures can be thought of as subprograms that are embedded in the main program. The statement

```
write('ABCD');
```

is a procedure call statement that invokes the *write* procedure. The *write* procedure is predefined; that is, it is automatically available to all IP programs. In the example, the procedure call statement passes the string ABCD to the *write* procedure as a **parameter**; *write* will display the string on the screen. The syntax for a procedure call is shown in Figure 5-5.

Parameter: A variable, or the current value of a variable that is passed to a procedure or function.

Figure 5-5. Procedure Call Syntax



Function calls are not classified as statements, because they express a value that can appear as an operand in an expression.

The procedure identifier is the name of the procedure to be called. The list of expressions in parentheses is called the parameter list of the procedure call. The number of parameters in the list depends on the procedure being called.

See Chapter 6 for complete information on procedures.

The effect of a procedure call is to execute the procedure immediately (passing the specified parameters, if any). When the procedure terminates, control is transferred to the statement following the procedure call.

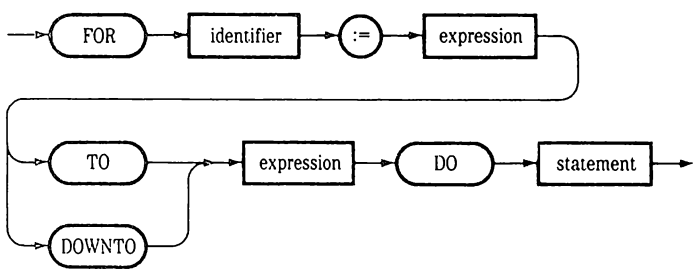
The Repetition Statements

Pascal has three statements that can repeatedly execute a statement or sequence of statements. Termination of the repetition is determined by the state of control variables or expressions. The repetition statements are the FOR, REPEAT, and WHILE statements.

The FOR Statement

The FOR statement is used to execute a statement a specific number of times. This is done by executing the statement once for each value of a control variable between an initial value and a limit value. The syntax of the FOR statement is shown in Figure 5-6.

Figure 5-6. FOR Statement Syntax



In this diagram, the identifier is the identifier of a variable called the **control variable**. The control variable may be of any ordinal type. The two expressions must be of the same ordinal type as the control variable.

Important

The control variable must be a simple variable; it cannot be an array element, a record field, or a dynamic variable.

The value of the initial expression is called the **initial value**, and the value of the limit expression is called the **limit value**.

For example, suppose that we have an array of 24 integer values and an integer value that can be used to index it:

```
VAR
 Val: ARRAY [1..24] OF integer;
 IX: INTEGER;
```

Now suppose that you want to multiply each value in the array by 2. You can do this with a FOR statement:

```
FOR IX := 1 TO 24 DO
 Val[IX] := 2*Val[IX]
```

Note that the control variable can be accessed within the FOR statement. (Remember that the control variable, like any other variable, must be declared. Also, it must be declared in the innermost block containing the FOR statement.) However, the value of the control variable must not be changed within the FOR statement.

Important

It is an error if the value of the control variable is changed by the execution of the FOR statement.

After the FOR statement is executed, the value of the control variable is undefined, unless the FOR statement was terminated by a GOTO.

In the example, the embedded assignment statement is executed 24 times. Each time, the control variable (IX) takes on a new value; this value is 1 the first time and 24 the last time.

If we want to write out each value after multiplying by 2, we can use a compound statement inside the FOR statement:

```
FOR IX := 1 TO 24 DO
 BEGIN
  Val[IX] := 2*Val[IX];
  writeln(Val[IX])
 END
```

Processing of the FOR statement using TO is as follows:

- First, the initial value is calculated (just once) and assigned to the control variable.
- Then, the limit value is calculated (just once).

If the initial value is greater than the limit value, the remainder of the FOR statement is skipped. Otherwise, the following steps are taken:

1. The statement following the word DO is executed.
2. The control variable is assigned the value of its own successor.
3. If the new value of the control variable is not greater than the limit value, go back to Step 1 and repeat. Repetition continues until the value of the control variable is greater than the limit value.

The DOWNTO option of the FOR statement sets the control variable to its own **predecessor** after each iteration, stopping when the value of the control variable is less than the limit value.

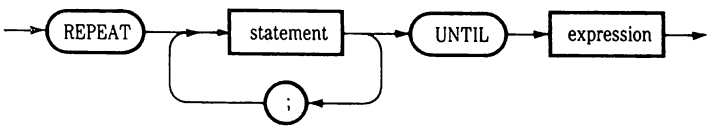
Some points to remember about Pascal FOR statements:

- A variable of any ordinal type may be used as a control variable.
- If the value of the control variable is changed by the execution of the contained statement, an error results.
- Use of the control variable in either limit expression produces undefined results.

The REPEAT Statement

The REPEAT statement, like the FOR statement, is used to control repetition in a program. The syntax is shown in Figure 5-7.

Figure 5-7. REPEAT Statement Syntax



Note the differences from the FOR statement. The sequence of statements in a REPEAT statement doesn't need to be delimited by a BEGIN and an END; the REPEAT and UNTIL do this. The expression after UNTIL must have a boolean result. It is evaluated after each execution of the enclosed statements; hence the statements are always executed at least once. Notice that if the expression is never *true*, the statements will be repeated forever.

A typical application of the REPEAT statement is

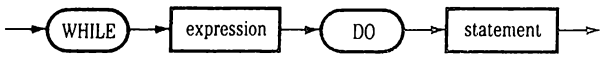
```
REPEAT
 write('Enter a number between 0 and 100 -> ');
 readln(IntVar);
 writeln('2 times ', IntVar, ' is ', 2 * IntVar)
UNTIL (IntVar < 0) OR (IntVar > 100)
```

This REPEAT statement executes the contained statements until the user types a number less than 0 or greater than 100.

The WHILE Statement

The WHILE statement is similar to the REPEAT statement. The syntax is shown in Figure 5-8.

Figure 5-8. WHILE Statement Syntax



Unlike the REPEAT statement, the WHILE statement evaluates the controlling expression before each repetition of its statement. If the expression, which must have a boolean result, is initially false, the statement will not be executed.

The WHILE statement acts on a single statement; thus a compound statement must be used if more than one statement is to be executed following the word DO.

For example, suppose that a program requires the user to type a number from 1 to 8. If the user does this, the program can continue. But if the user types a number that is out of range, the program must display an error message and give the user another chance; and this should be repeated until the user types a number in the required range. This is a natural application for the WHILE statement:

```
write('Type a number from 1 to 8: ');
readln(Number);
WHILE (Number < 1) OR (Number > 8) DO
 BEGIN
  writeln('Number must not be less than 1.');
  writeln('Or more than 8!');
  write('Try again.  Type a number from 1 to 8:  ');
  readln(Number)
 END
```

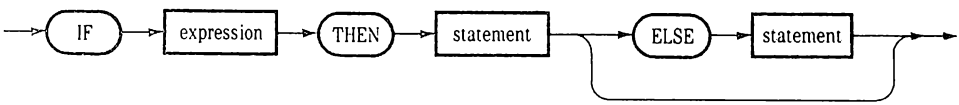
If the user gives a correct response the first time, the statement contained by the WHILE statement is never executed. But if the user's response is out of range, the WHILE statement executes repeatedly until the user responds correctly. In either case, the value of Number is guaranteed to be in the range 1..8 at the end of this sequence.

The IF statement and the CASE statement are used to execute a statement if a variable or expression has a desired value.

The IF Statement

The IF statement contains a boolean expression, a statement to be executed if the value of the expression is *true*, and (optionally) another statement to be executed if the value of the expression is *false*. The syntax of the IF statement is shown in Figure 5-9.

Figure 5-9. IF Statement Syntax



When an IF statement is executed, the following sequence of events takes place:

- First, the boolean expression is evaluated.
- If the boolean expression is *true* the statement following THEN is executed.
- On the other hand, if the boolean expression is *false* and there is an ELSE, the statement following ELSE is executed.
- If the boolean expression is *false* and there is no ELSE, then execution proceeds from the next statement following the IF statement.

Note that just one statement is allowed after the word THEN; it may be a compound statement. Likewise, just one statement (possibly compound) is allowed after the word ELSE. Here is an example of an IF statement without an ELSE part:

```
IF Total > 100 THEN
 BEGIN
  writeln('Error:  Total too big');
  Total := Total - Current
 END
```

If the value of the variable Total is greater than 100, the compound statement is executed to display the message `Error: Total too big` and adjust the value of Total; otherwise the compound statement is not executed.

The ELSE part of an IF statement is only executed if the result of the boolean expression is *false*. For example, the statement

```
IF Total > 100 THEN
 BEGIN
  writeln('Error:  Total too big');
  Total := Total - Current
 END
ELSE
 WRITELN('Total is ', Total)
```

will execute the compound statement if the value of Total is greater than 100 (just as in the previous example); but if the value of Total is not greater than 100, then the *writeln* statement following the word ELSE is executed to display the message `Total is` followed by the value of Total.

Important

Be careful of semicolon placement in IF statements. For example

```
IF A = B THEN
 BEGIN
  writeln('A equals B');
  EqualCount := EqualCount + 1
 END
ELSE
 writeln('A not equal to B')
```

is correct: there is no semicolon after the END because the ELSE does not start a new statement—it is a continuation of the same IF statement. A common mistake is to put a semicolon before the ELSE, which is a syntax error because there is no Pascal statement that begins with the word ELSE.

Nested IF Statements

The statement following the word ELSE can be an IF statement, and can contain its own ELSE clause. Thus a statement can be written to take different actions for each of several mutually exclusive conditions.

```
REPEAT
 write('Enter command S,D,P,Q,E -> ');
 readln(Command);
 IF Command = 'S' THEN
  ShuffleDeck
 ELSE IF Command = 'D' THEN
  DealCards
 ELSE IF Command = 'P' THEN
  DisplayPoints
 ELSE IF (Command = 'Q') OR (Comm = 'E') THEN
  Quit
UNTIL (Command ='Q') OR (Command ='E')
```

Conditions will be checked only until a true one is found. The greatest efficiency is achieved if the most probable conditions are checked first.

The statement following the word THEN can also be a nested IF statement, but this construction is less useful and can lead to a confusing program. Be careful with the following type of construction:

```
IF A=B THEN
   IF C=D THEN
    writeln('A=B and C=D')
   ELSE
    writeln('A=B but C<>D')
```

The ELSE matches the last preceding THEN, as indicated by the indentation. If you add another ELSE, it will match the first THEN:

```
IF A=B THEN
 IF C=D THEN
  writeln('A=B and C=D')
 ELSE
  writeln('A=B but C<>D')
ELSE
 writeln('A<>B')
```

This statement can be clarified, without changing its meaning, by making the nested statement a compound statement as shown in the following example.

```
IF A=B THEN
 BEGIN
  IF C=D THEN writeln('A=B and C=D')
  ELSE writeln('A=B but C<>D')
 END
ELSE
 writeln('A<>B')
```

The CASE Statement

The CASE statement uses the value of an expression to select and execute one statement from a list of statements. The controlling expression and the list of statements are contained within the CASE statement, and each statement in the list is labeled with one or more constants called **case constants**, which are possible values of the controlling expression. An OTHERWISE clause is optional; if present, it contains a statement that is executed if the controlling expression's value does not match any of the case selectors.

The syntax of the CASE statement and a CASE clause are shown in Figures 5-10 and 5-11.

Figure 5-10. CASE Statement Syntax

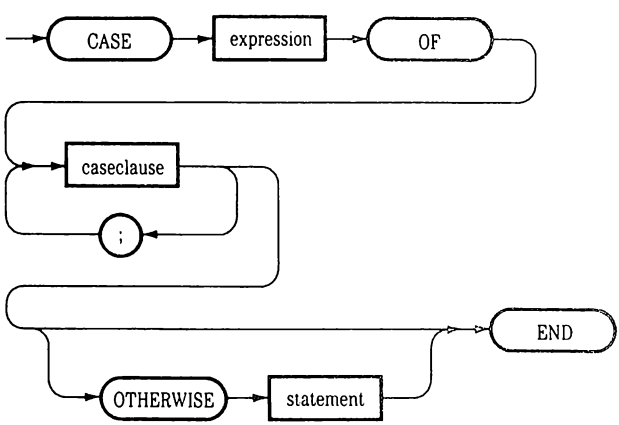
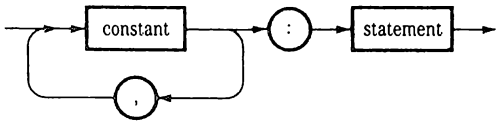


Figure 5-11. CASE Clause Syntax



In these diagrams, the expression must give a result of an ordinal type, and the constants in each CASE clause must be of the same type. The expression is evaluated, and the result is sequentially compared with the constants in each CASE clause. If the result matches one of the constants, only the statement in that CASE clause is executed.

If no match is found and there is an OTHERWISE clause, the statement in the OTHERWISE clause is executed. If no match is found and there is no OTHERWISE clause, an error occurs. An empty OTHERWISE clause will prevent this from occurring, by causing execution to proceed to the next statement after the CASE statement.

This example was used in the discussion of nested IF statements:

```
REPEAT
 write('Enter command S,D,P,Q,E -> ');
 readln(Command);
 IF Command = 'S' THEN
  ShuffleDeck
 ELSE IF Command = 'D' THEN
  DealCards
 ELSE IF Command = 'P' THEN
  DisplayPoints
 ELSE IF (Command = 'Q') OR (Command = 'E') THEN
  Quit
UNTIL (Command='Q') OR (Command='E')
```

Exactly the same effect can be achieved more naturally with a CASE statement:

```
REPEAT
 write('Enter command S,D,P,Q,E -> ');
 readln(Command);
 CASE Command OF
  'S':  ShuffleDeck;
  'D':  DealCards;
  'P':  DisplayPoints;
  'Q', 'E':  Quit
 END
UNTIL (Command='Q') OR (Command='E')
```

If you use the nested IF statement and try to allow for the possibility of lowercase letters, the resulting nest would be unwieldy. With a CASE statement, however, the enhancement is easy. You can also add an OTHERWISE clause to handle invalid command input by calling a procedure named HELP.

```
REPEAT
 write('Enter command S,D,P,Q,E -> ');
 readln(Command);
  CASE Command OF
    'S', 's': ShuffleDeck;
    'D', 'd': DealCards;
    'P', 'p': DisplayPoints;
    'Q', 'E', 'q', 'e': Quit;
 OTHERWISE Help
 END
UNTIL (Command='Q') OR (Command='E')
 OR (Command='q') OR (Command='e')
```

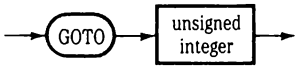
As in all other cases where only a single statement is allowed, each statement within a CASE clause or OTHERWISE clause can be a compound statement.

The GOTO Statement

Some programming situations demand an instant transfer of control in a manner that is not easy to achieve using the repetition or conditional statements. To handle these situations, Pascal provides the GOTO statement. The GOTO statement should be used only in those unusual cases that cannot be handled easily by the other control statements.

The GOTO statement causes a direct transfer of control to a labeled statement that is in the same block as the GOTO statement (considering the main program to be a procedure). The syntax of the GOTO statement is shown in Figure 5-12.

Figure 5-12. GOTO Statement Syntax



The label must be an unsigned integer of not more than four digits. The label must first be declared. Label declarations come immediately after the heading of a program, procedure, or function, before any other declarations. The following program, which loops infinitely, shows legal use of label declarations and the GOTO statement. It also shows some of the problems of the GOTO statement.

```
PROGRAM Jump;
 LABEL
   1, 5326, 42, 999;
BEGIN
1:
  GOTO 5326;
999:
  GOTO 42;
5326:
  GOTO 999;
42:
  GOTO 1
END.
```

Important

When the destination of a GOTO is in a block that does not contain the GOTO, every block **activation** that has occurred since the most recent activation of the current block is terminated.

Also, remember that the constants that introduce cases within a CASE statement are not labels and cannot be referenced in GOTO statements.

Activation: The execution of a block of code. Normally a block will have either no activations (if it is not currently being executed) or one activation (if it is being executed). A block that is recursive may have more than one activation at a time.

Passing control to a statement that is inside a structured statement from a point outside the structured statement by means of a GOTO statement causes an error to occur. Be extremely careful when using GOTO statements. All of the GOTO statements in the following example are wrong for this reason.

```
IF true THEN
123:
  GOTO 6;
 FOR Index := 1 TO 10 DO
6:
  GOTO 123;
 BEGIN
1:
  writeln('GOTO another procedure');
  GOTO 6
 END;
 GOTO 1
```

When the destination statement of a GOTO is in a block that does not contain the GOTO, the destination statement must be at the outermost statement level. For example:

```
Procedure X;
BEGIN
IF true THEN
 123: writeln('You can't get here from there.');
END;
BEGIN
 GOTO 123;  {This is illegal.}
END;
```

In this variation, the destination statement is at the outermost level:

```
Procedure Z;
BEGIN
123: writeln('You can get here from there.');
END;
BEGIN
 GOTO 123;  {A legal GOTO}
END;
```

The WITH Statement

The WITH statement is used to refer to the fields within a record variable, without having to write the identifier of the record repeatedly. This statement is discussed in detail in Chapter 8, "Records."

Chapter 6

Procedures and Functions

Procedures and **functions** are the subroutines of Pascal. Each procedure or function is a distinct section of code, contained within a program, that is executed when the program calls it. In many cases, the program calls it more than once.

A procedure can be thought of as a subprogram nested in the main program (or within another procedure, or within a function). Just as you define a Pascal program by writing it in text form, you define a procedure by writing a **procedure declaration** into the text of the program. If you study the syntax diagrams further on in this chapter, you can see that a procedure declaration, like a program, contains one block. The block may contain other procedure declarations. Thus procedures (and functions) can be freely nested within each other. Indeed, for purposes of program execution the system considers the program itself to be just the outermost procedure of a nested structure of procedures and functions.

Blocks are defined in Chapter 2, "Program Components."

A procedure is called by means of a procedure call statement, which refers to the procedure by name and supplies values for any **parameters** belonging to the procedure. Parameters are a special kind of variable used to pass information to the procedure when it is called; they are discussed in detail below.

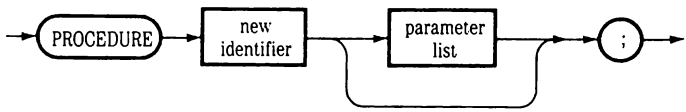
A function is similar to a procedure except that it is called by means of a function reference instead of a call statement. The function reference appears in an expression; it references the function by name and supplies any parameters required by the function. The function returns a value; that is, it computes a value, and this value replaces the function reference when the expression is evaluated.

Procedures and functions are declared (written) after the variable declarations, if any, and before the compound statement that contains the statements of the block.

Declaring a Procedure

A procedure declaration consists of a procedure heading, a block, and a terminating semicolon. Procedure declaration syntax is shown in Figure 6-1.

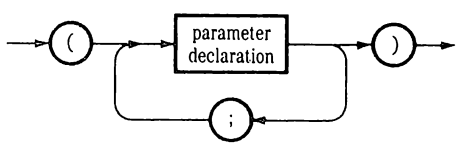
Figure 6-1. Procedure Declaration Syntax



Formal parameter: In the declaration of a procedure, the parameter that will be used to pass information into the procedure for processing.

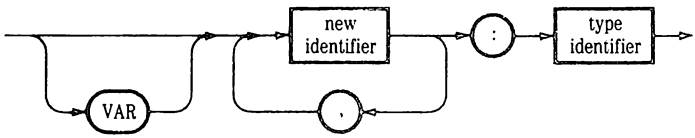
The first part of the procedure declaration—the word PROCEDURE, the identifier, and the **formal parameter** list—are called the **procedure heading**. A semicolon follows the procedure heading and separates the heading from the **procedure body**. The procedure body consists of a block and a final semicolon. The syntax for a parameter list is shown in Figure 6-2.

Figure 6-2. Parameter List Syntax



The formal parameter list declares the procedure's formal parameters, if any. It consists of an opening parenthesis, one or more formal parameter declarations separated by semicolons, and a closing parenthesis. The syntax for each formal parameter declaration is shown in Figure 6-3.

Figure 6-3. Parameter Declaration Syntax



If the word VAR is used, this declaration declares one or more **variable parameters**; otherwise it declares one or more **value parameters**. The distinction is explained below. Each declaration can declare any number of parameters, all of the same type. Note that the type must be given as a single identifier such as *real*, *char*, or the identifier of a type that has been declared in the program. This is one of the reasons for declaring types.

Important

If the type of a formal parameter is type STRING, then any STRING type is considered identical to it; the size attribute of the formal parameter is always the size attribute of the actual parameter.

Here is an example of a simple procedure heading that declares three value parameters:

```
PROCEDURE Alpha (Initial, Limit:  extended;
        Count:  integer);
```

Initial and Limit are *extended* parameters, and Count is an *integer* parameter. The following example declares a variable parameter and a value parameter:

```
PROCEDURE Beta (VAR ErrorFlag:  boolean;
        N:  integer);
```

ErrorFlag is a variable parameter of type *boolean*, and N is a value parameter of type *integer*.

The rest of a procedure definition consists of one block, as described in Chapter 2. A block consists of optional declarations, optional procedure and function definitions, and one compound statement. Further on in this chapter, you will see a special case where the block is replaced by the word FORWARD.

Value Parameters

Value parameter: A parameter that is passed to a procedure by value, rather than by address.

Variable parameter: A parameter that is passed to a procedure by address, rather than by value.

There are two kinds of parameters: **value parameters** and **variable parameters**. Every parameter is a value parameter unless it is explicitly declared as a variable parameter (see the next section). Value parameters are used to pass values (of expressions or variables) to a procedure or function at the time it is called.

The following example shows how value parameters can be used:

```
PROCEDURE WriteMean (A, B:  extended);
 VAR
  Sum:  extended;
BEGIN
 Sum := A + B;
 writeln(Sum/2)
END;
```

WriteMean has two **formal parameters**, A and B; both are value parameters of type *extended*. A and B are, in effect, *extended* variables belonging to the WriteMean procedure; but they have the special property that each time the procedure is called, A and B are given the values of **actual parameters** contained in the procedure call statement. The call statement must provide an actual parameter for each formal parameter. The types of the actual and formal parameters must match.

The actual parameters are expressions. Each expression is evaluated and the result is assigned to the corresponding formal parameter before the statements of WriteMean are executed.

All of the following are valid calls to WriteMean (with different results):

```
WriteMean(4.3, X)
WriteMean(X, Y)
WriteMean(Z + 2.3*Y, X)
WriteMean(25, Z)
```

In these procedure calls, assume that X, Y, and Z are numeric variables or constants. Note that each actual parameter in the procedure call is an expression; the expression is evaluated, and the value is passed to the procedure. The fourth example shows that an integer value may be supplied for a real-type parameter; the integer value is converted to a real-type value just as if it were being assigned to a real-type variable.

Important

The type of a value parameter can be any Pascal data type except a file type or of any structured type that contains a file type. To pass a file to a procedure or function, you must use a variable parameter (see next section).

Variable Parameters

A value parameter, as we have seen, provides one-way communication between the calling program and the procedure or function: the call supplies a value, and this value is used inside the procedure or function. A variable parameter provides two-way communication.

With a variable parameter, the actual parameter is not an expression but a variable reference, and the information passed to the procedure or function is not the value of the variable but the variable itself. Note that this variable is one declared outside the procedure or function.

The type of the actual parameter must be identical to that of the formal variable parameter. However, if the parameter type is STRING, then any string type is considered identical to it; the size attribute of the formal parameter is always the size attribute of the actual parameter.

Important

When the program is executed, it is the address of the actual parameter that is passed to the procedure, so that the code of the procedure or function can access it.

If the value of the formal parameter is changed inside the procedure or function, the effect is to change the value of the actual parameter variable (outside the procedure or function). The declaration of the formal parameter is preceded by the reserved word VAR, as in the following example:

```
PROCEDURE Movit(Rho, Theta:  extended; VAR X, Y:  real);
{Update rectangular coordinates X and Y for motion}
{through distance Rho at angle Theta (in degrees).}
CONST
 Pi: 3.17159;

BEGIN
 Theta := Theta*Pi/180;                    {Convert to radians}
 X := X + (Rho * cos(Theta));
 Y := Y + (Rho * sin(Theta));
END;
```

The *cos* and *sin* functions are predefined by Instant Pascal. They assume that angles are given in radians. The Movit procedure has two value parameters, Rho and Theta, and two variable parameters, X and Y. Suppose that the Movit procedure is called by the statement

```
Movit(Radius, Angle, Horiz, Vert)
```

where Radius, Angle, Horiz, and Vert are *real* variables. Movit assigns the current values of Radius and Angle to its own value parameters Rho and Theta, respectively. It also associates variables X and Y with the identifiers Horiz and Vert—in effect assigning to X and Y whatever values are currently held by Horiz and Vert.

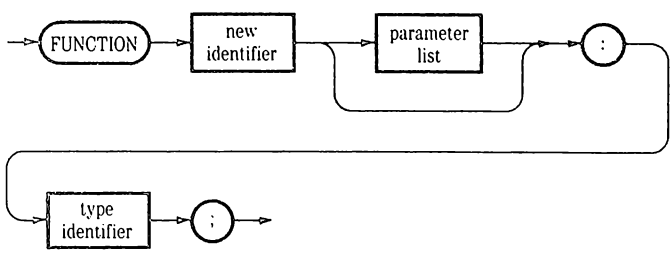
When Movit executes, it converts the value of Theta to radians without affecting the variable Angle. But when it changes the values of X and Y, it changes the values of Horiz and Vert. Because Movit was called with Horiz and Vert as actual parameters for the formal variable parameters X and Y, each reference to X during this execution of Movit is in effect a reference to Horiz; and each reference to Y is in effect a reference to Vert.

Of course, Movit could have been written with direct reference to Horiz and Vert, instead of X and Y. But that would make it less flexible; by using variable parameters, Movit is able to update any two *real* variables that are passed to it.

Declaring a Function

Syntactically, a function declaration is very similar to a procedure declaration. The heading has the word FUNCTION instead of PROCEDURE, and it specifies a type, which is the type of the value returned by the function. Function declaration syntax is shown in Figure 6-4.

Figure 6-4. Function Declaration Syntax



Within the function, there should be an assignment statement that has the function identifier on the left side. This is how the function returns a value. For example, consider the WriteMean procedure shown earlier. It displays a result on the screen. For some programs, it would be more useful to declare a function that would perform the same calculation and return the result:

```
FUNCTION Mean (A, B:  extended):  extended;
BEGIN
 Mean := (A + B)/2
END;
```

The value of (A + B)/2 will be a result of type *extended*. The value of the function is declared to be type *extended* to ensure accuracy. If the value of the function were another of the real-types the *extended* result would be rounded before being returned.

If no value is assigned to the function identifier, an error will occur. If there are two or more assignment statements with the function identifier on the left side, the last value assigned as the function executes is the value returned.

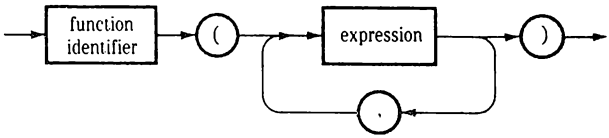
Important

Normally the function identifier should be used within the function only as shown in the example just given—on the left side of an assignment statement, for the purpose of returning a value. Do not use the function identifier on the right side of an assignment statement within the function, unless the function is designed to be recursive. Recursive functions and procedures are discussed in this chapter in the section titled "Recursion."

Calling a Function

As mentioned in Chapter 2, a function is activated by a function call, which appears as an operand in an expression. When the operand is evaluated at run time, the function is executed. The value returned by the function becomes the value of the operand. The syntax of a function call is shown in Figure 6-5.

Figure 6-5. Function Call Syntax



Recursion

A recursive procedure or function is one that calls itself; this is permitted in all Pascal procedures and functions. A full discussion of the idea of recursion is beyond the scope of this manual; instead, a single example is offered as an illustration.

Consider the following (hypothetical) situation: A 30 x 50 array of type Color is used to represent a graphic picture 30 dots high and 50 dots wide. Each value in the array represents a dot in the picture—each dot is either White or Black. The array is declared as follows:

```
TYPE
 Color = (Black, White);
VAR
 Pic:  ARRAY[1..30, 1..50] OF Color;
```

Now suppose that the picture contains several images, each of which consists of a group of white dots that are connected to each other—that is, every dot in an image is a neighbor of at least one other dot in the image (horizontally, vertically, or diagonally). In terms of the array, this means that an image is a collection of White elements, and each of these elements is a neighbor of at least one other element in the image. One array element is a neighbor of another if their indices differ by 1 or 0, and both have the value White.

The problem is this: how to write a procedure called Zap that will erase (that is, change the value of a dot from White to Black) all the dots in one image, if it is given the coordinates (indices) of any one dot in the image—without affecting any other image in the picture. This is an unwieldy problem if Zap has to find all the dots in the image in a single pass; but if Zap can call itself, the problem becomes simple.

In English, the Zap procedure is

"IF the specified dot is in the array AND it is white, THEN erase it and Zap each of its neighbors in turn. ELSE do nothing and return immediately."

When Zap is called once with the indices of a dot in an image, it will eventually call itself for every other dot in the image. To write Zap in Pascal, the first step is to write a convenient function for checking that a pair of indices is valid—that is, that both indices are within the bounds of the array Pic:

```
FUNCTION InArray(I, J:  integer):  boolean;
BEGIN
 InArray := (I IN [1..30]) AND (J IN [1..50])
END;
```

The reference InArray(A, B), where A and B are integer values, will return *true* if A and B are both valid indices for Pic. Now we can write Zap as follows:

```
PROCEDURE Zap(X,Y:  integer);

VAR
 XN, YN:  integer;                          {Two variables to be used}
                                            {as coordinates of neighbors.}

BEGIN

 IF InArray(X,Y) THEN                       {If X,Y is in the array}
  IF Pic[X,Y] = White THEN                  {and is a white dot...}
   BEGIN
    Pic[X,Y] := Black;                      {...then erase it...  }
    FOR XN := X-1 TO X+1 DO                 {...and ZAP all its neighbors:}
     FOR YN := Y-1 TO Y+1 DO
      Zap(XN,YN)
   END
END;
```

Important Notice that in the process of Zapping all the neighbors, Zap will also Zap the dot that it started with. This is harmless, because the dot is no longer white; this particular recursive call will do nothing and return immediately.

Each time a recursive routine calls itself, a new incarnation of the routine is created—that is, all the data belonging to the current incarnation has to be saved and new space allocated for the data belonging to the new

incarnation. Eventually, if the routine is written correctly, the recursion terminates; the last incarnation does not call itself, but simply returns to the previous incarnation, which returns to the one before it, and so forth. Finally the first incarnation returns, and the execution of the recursive routine is finished.

In the case of Zap, each incarnation makes 9 recursive calls in sequence. Each of these calls starts a new chain of incarnations. Each chain terminates when the dot that it is supposed to Zap turns out not to be in the array, or not to be a white dot. When all the dots in the image have been erased, then all the chains have terminated, and all the incarnations have returned; the original incarnation returns to the point in the program where Zap was called nonrecursively.

Termination

In order to make sense, a recursive function or procedure has to be written so that it will always terminate. This means that there is some condition under which it will not call itself; furthermore, if it calls itself enough times, it will always arrive at that condition. Otherwise, it may keep calling itself until the system runs out of space to keep track of all the recursive calls, and halts the program with an error message telling you you're out of memory. Actually this can happen even if the recursive procedure or function is correctly written, because of the space required by the numerous incarnations. When this happens, some sort of rewriting is required. A full discussion of space-saving techniques is beyond the scope of this manual, but the following suggestions may prove helpful:

- Find a way to make it terminate sooner.
- Reduce the amount of storage used for variables inside the recursive routine, since this storage has to be replicated for each recursive call.
- Use files to store large data structures on disk instead of in memory (see Chapter 10).

Indirect Recursion

The above discussion describes direct recursion; there is also such a thing as indirect recursion. Suppose that a program contains three or more procedures or functions called A, B, C, and so forth. If A calls B and B calls A, that is indirect recursion. Likewise, if A calls B, B calls C, and C calls A, we have indirect recursion. The most general definition of recursion is that it occurs whenever a procedure or function is called (by itself or by another procedure or function) before it completes its execution and returns.

Like direct recursion, indirect recursion requires that there be a condition that will terminate the recursion, and the procedures must be designed to guarantee that the termination condition will be reached.

Frequently, when you write indirectly recursive procedures or functions, one procedure or function must reference another procedure or function before it has been formally declared. This is impossible using normal procedure or function definitions, so Pascal provides a special form of definition called the **forward definition**. In a forward definition the block is replaced by the word FORWARD. The forward definition suffices to declare the identifier and parameters, and the type in the case of a function definition. The forward-defined procedure or function can then be called in a following procedure or function, and then the remainder of the forward definition can be given as in the following example (where function F calls procedure P and vice versa):

FORWARD is not an IP reserved word—it's a directive. Unlike reserved words, the meaning of a directive can be changed by a new declaration. However, this is a practice that should be unnecessary and is best avoided.

```
{Forward definition of F, to allow it to be referenced within P:}
FUNCTION F (X, Y:  extended; Count:  integer):  extended;
FORWARD;

{Normal definition of P, which calls F:}
PROCEDURE P (N: integer);
 VAR
  A, B, C: real;
BEGIN
  .
  .
  .                        {Various statements}
  C := 2 *  F(A, B, N)     {This calls F}
  .                        {Various statements}
  .
  .
END;

{Continued definition of F; parameters and type omitted}
{since they are already declared:}
FUNCTION F;
 VAR
  TMP, DL, DX, DY: extended;
BEGIN
  .
  .
  .                        {Various statements}
  P(trunc(X))              {This calls P}
  .
  .
  .                        {Various statements}
END;
```

Chapter 7

Arrays, Sets, and Strings

Up to this point, all the data types discussed have been simple data types, which have single values. Pascal also has a variety of **structured data types**, which can be thought of as collections of values. This chapter covers only the three simplest kinds: arrays, sets, and strings. Subsequent chapters cover records and files.

An **array** is a collection of objects called **components**; all the components of an array are of the same type. A single component of the array is referenced by using the array identifier with one or more **index values** (sometimes called **subscripts**). The index values select the desired component from among the other components of the array.

A **set** is a collection of values that are called **members** of the set. Set operations allow very straightforward coding of routines that would be much more complicated without the use of sets.

A **string** is a sequence of characters, normally treated as a single entity. Instant Pascal provides several predefined functions for manipulating strings.

Arrays

An array is a collection of components, all of the same type. Each component in an array can be considered as a variable in its own right. A particular component is distinguished from other components by means of one or more index values enclosed in square brackets. For example, you can declare an array called XYZ, consisting of three components of type *real* numbered 1, 2, and 3, as follows:

```
VAR
 XYZ:  ARRAY [1..3] OF real;
```

Then these components can be referred to individually as XYZ[1], XYZ[2], and XYZ[3]. Each of them is a variable of type *real*.

Pascal arrays differ from arrays in other languages in several ways:

- Pascal arrays can have any number of dimensions.
- The components of a Pascal array can be of any type.
- The values used to index components of a Pascal array can be of any ordinal type. This means that the first component of an array is not necessarily component 0, or component 1; it depends on how the array is declared.

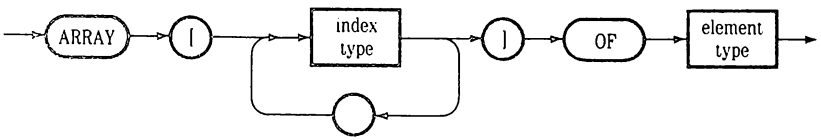
An array can be treated as a unit, without indexing, in three ways:

- It can be passed to a procedure or function as an actual parameter, if its type is identical to the type of the formal parameter.
- It can be assigned to another array of identical type.
- It can be compared to another array of identical type.

Appendix B describes the rules governing type compatibility.

The syntax of an array type is shown in Figure 7-1.

Figure 7-1. Array Type Syntax



The examples in this chapter use variables of these real-types: *real*, *double*, and *extended*. These are described in both Chapter 3 and Appendix E.

Note that there can be more than one index type: one for each dimension of the array type. First we will consider one-dimensional arrays.

One-Dimensional Arrays

An array type can be used in a type declaration as follows:

```
TYPE
 Values = ARRAY [0..99] OF extended;
```

or in a variable declaration as follows:

```
VAR
 Values:  Array [0..99] OF extended;
```

The index type is usually a subrange of type *integer*, but it can also be any other ordinal or subrange type. The component type can be any type.

The index type determines the number of components in the array: there is one component for each possible value of the index. For example, consider the following declaration:

```
VAR
 TenReals:  ARRAY [0..9] OF real;
```

The index type is the subrange 0..9, so the array TenReals will have 10 components, each one of which is of type *real*. The first component is TenReals[0], the next is TenReals[1], and so on; the last component is TenReals[9].

Multidimensional Arrays

Because the type of the components can be anything except a file type, you can declare an array of arrays. For example, here is a declaration of an array whose components are arrays like the one declared above:

```
VAR
 Square:  ARRAY [0..9] OF ARRAY [0..9] OF double;
```

Square has ten components, each of which is an array of ten *real* values. The variable Square[3] is an array variable, and you can think of Square as a 10 x 10 matrix of *double* values; Square[3] can be thought of as a row, and its *double* components can be thought of as the column components of the row. To select one of the *double* values from the matrix, you need two indices; for example, Square[6][5] refers to row 6, column 5.

Thinking of the first index as a row index and the second as a column index is a matter of choice; you could just as well think of the first index as a column index and the second as a row index.

Instead of writing Square[6][5], you can write both indices in one pair of brackets, with a comma to separate them: Square[6,5]. The two notations are equivalent and interchangeable. Similarly, you can condense the declaration of Square by writing

```
VAR
 Square:  ARRAY [0..9, 0..9] OF double;
```

This declaration means exactly the same thing as the previous one, and has the advantage of being more explicit to the human reader. It obviously declares a two-dimensional array of *double* values. Here is a declaration of a three-dimensional array:

```
VAR
 Space:  ARRAY [0..MaxX, 0..MaxY, 0..MaxZ] OF real;
```

where MaxX, MaxY, and MaxZ are previously declared *integer* constants. A Pascal array can have as many dimensions as desired.

Other Index Types

So far, all the examples have shown integer subranges as index types, since this is the most common usage. However, remember that an index type can be of any ordinal type. For example, it can be *char*, or a subrange of *char*.

The declaration

```
VAR
 Crypt:  ARRAY [char] OF char;
```

creates an array of characters that is indexed by character values. It has one component for each possible character value, or 256 components in all. Such an array could be useful for a cryptography routine.

The indices of a multidimensional array can be of different types. For a more complicated cryptographic scheme, you might declare

```
VAR
 CryptArray:  ARRAY [char, 1..KeyMax] OF char;
```

where KeyMax is a previously declared constant. CryptArray contains one component for each possible combination of a character value and an *integer* value from 1 to KeyMax.

Index Values

In a reference to a specific component of an array, each index value is given as an expression. For example, the following is a valid assignment statement:

```
Space[X,Y,Z] := Space[X-DX, Y-DY, K];
```

The only restriction on an expression used as an index value is that the type of the expression's value must be compatible with the index type in the array declaration; if the index type is a subrange, the index value must be within the subrange.

Passing Arrays

A procedure or function parameter (either value or variable) can be declared to be an array; then when the procedure or function is called, an array of an identical type can be passed as the actual parameter.

Array Assignments

An array can be assigned to another array of an identical type. For example, if you have the declarations

```
VAR
 AAA, BBB:  ARRAY [1..255] OF integer;
```

then we can assign all the values of BBB to the corresponding components of AAA as follows:

```
AAA := BBB
```

Array Comparisons

An array can be compared with another array of identical type. The only comparison operators allowed for arrays are the = and <> operators . For example, if you have the declarations

```
VAR
 CCC, DDD:  ARRAY [0..99] OF real;
```

then you can compare CCC to DDD as follows:

```
CCC = DDD
```

The result of this expression is *true* if every component of CCC has the same value as the corresponding component of DDD. The other possible comparison is

```
CCC<>DDD
```

which is *true* if any component of CCC has a different value from the corresponding component of DDD.

Packed Arrays

Instant Pascal provides the ability to declare variables of type PACKED ARRAY, and the *pack* and *unpack* procedures are provided for compatibility with other ANSI Pascal systems. The compression of data in memory that is usually done with these procedures isn't compatible with the way in which data is stored in an Apple II. However, if you write IP programs with the intention of transporting them to a system that uses packed data structures, you can include these procedures.

The *pack* and *unpack* procedures transfer data between PACKED and unpacked arrays.

The Pack Procedure

The *pack* procedure takes the following form:

pack(a, i, z)

Parameter a is a variable of an array type and z is a variable of a packed array type whose component type is the same as the component type of a. Parameter i is an expression whose value is assignment compatible with the index type of a.

The *pack* procedure transfers the components of array a to array z, beginning with the component number specified by i.

For example, the statement

```
pack(NewNames, 3, NewPNames);
```

transfers the components of NewNames (an array) to NewPNames (a PACKED array), beginning with the third element.

Important

Note that the number of elements transferred is the number of the elements in the entire packed array. If there are not enough elements in the portion of the unpacked array specified by the component number, an error occurs.

The Unpack Procedure

The *unpack* procedure takes the following form:

unpack(z, i, a)

The *unpack* procedure is the opposite of the *pack*. procedure: it transfers the components of a PACKED array to an unpacked array. Given the example just mentioned, this statement:

```
unpack(NewPNames, 4, NewNames);
```

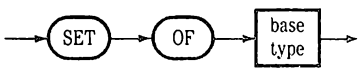
would transfer the components of NewPNames (a PACKED array) to array NewNames (an unpacked array), beginning with the fourth element.

As with the *pack* procedure, *unpack* transfers every element in the specified packed array. If there are not enough elements in the portion of the packed array specified by the component number, an error occurs.

Sets

In Pascal, a set is a collection of distinct values, all of the same ordinal type. The values are called **members** of the set; the type of the members is called the **base type** of the set. The syntax of a set type is shown in Figure 7-2.

Figure 7-2. Set Type Syntax



Here are some examples of set variable declarations:

```
VAR
 Letters, SpecialChars, PrintingChars:  SET OF char;
 Digits:  SET OF '0'..'9';
 Colors:  SET OF (Violet, Blue, Green, Yellow, Orange, Red);
```

A set type specifies all the possible members of a set of that type. For example, a SET OF *char* can contain any collection of distinct char values. The term "distinct" here means that a particular *char* value can only appear once in the set; for example, the set Digits, declared above, either contains the character "5" or it doesn't.

When the type given after the words SET OF is a subrange, the **host type** of the set is the host type of the subrange. Thus in the second example above (SET OF '0'..'9'), the host type of the set Digits is the char type; the set can contain only the characters 0 through 9.

Set Values

To understand set values, it may be helpful to know that a set value is represented internally as a bit pattern, with a bit for each possible member of the set. Each of these bits indicates whether that possible member is actually a member. The point of this is that although a set is a collection of members, a **set value** is a single value that represents some or all of the members of that set.

To write a set value explicitly, you specify its members between square brackets. For example, suppose that the variable SpecialChars has been declared as a SET OF *char* (as shown above). Now consider the assignment statement

```
SpecialChars := ['.', ',', ';', ':', '(', ')']
```

After this assignment, SpecialChars is the set containing the period, comma, semicolon, colon, and left and right parentheses. Internally, this value is represented by a bit pattern containing 256 bits, one for each possible *char* value. In this pattern, the particular bits corresponding to the characters that are members of the set are on and the other bits are off.

When the members form a subrange, you can use the subrange notation, as in

```
Digits := ['0'..'9']
```

This makes Digits the set of all characters from 0 through 9. Alternatively, if you wanted Digits to contain only octal digits, you could write

```
Digits := ['0'..'7']
```

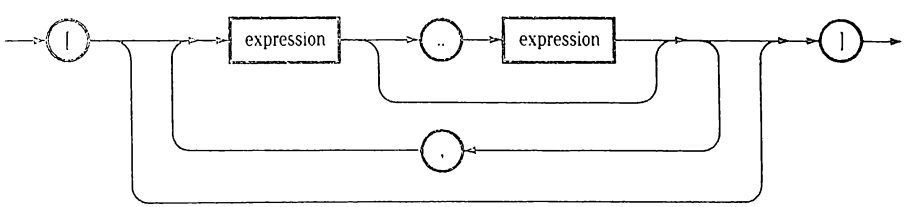
This makes Digits the set of all characters from 0 through 7. You can also use more than one subrange and mix subranges and individual members as in the following:

```
Letters := ['A'..'Z', 'a'..'z']
Colors := [Violet..Green, Orange];
```

After these two assignments, Letters is the set of all capital letters and all lowercase letters, and Colors is the set containing Violet, Blue, Green, and Orange.

A set value written with square brackets in this manner is called a **set constructor**. The syntax for a set constructor is shown in Figure 7-3.

Figure 7-3. Set Constructor Syntax



Restrictions on Sets

The host type of a set cannot have more than 256 values. A set cannot contain any value whose ordinality is less than 0 or greater than 255. In particular, it cannot contain any integer less than 0 or greater than 255. An attempt to assign more than 256 members to a set, or to assign an integer member outside the range 0..255, results in an error.

The IN Operator

The IN operator is used to test whether a particular ordinal value is a member of a particular set. The IN operator is a relational operator, and has a *boolean* result. It has the same precedence as the other relational operators.

The IN operator must have an ordinal value on the left and a set on the right. The type of the ordinal value must be the same as the base type of the set. Suppose that we have the following declarations and assignments:

```
TYPE Color = (Magenta, Cyan, Yellow);

VAR
 Letters:  SET OF char;
 Colors:  SET OF Color;
 InChar, TestChar:  char;
 Index:  integer;
 Tint:  Color;

BEGIN
 Letters := ['A'..'Z', 'a'..'z'];
 Colors := [Magenta, Cyan];
 .
 .
 .
```

Then the following expressions are valid uses of IN:

```
InChar IN Letters           {true if value of InChar is a letter}

TestChar IN ['a'..'z']      {true if value of TestChar}
                            {is a lowercase letter}

succ(Tint) IN Colors        {true if successor of value}
                            {of Tint is Cyan}
(Index + 5) IN [0..255]     {true if value of (Index +5)}
                            {is in the range -5..250}
```

All of these expressions could be replaced with constructs that do not use sets; for example, InChar IN Letters could be replaced with

```
(InChar >= 'A') AND (InChar <= 'Z') OR
(InChar >= 'a') AND (InChar <= 'z')
```

However, the expression "InChar IN Letters" is much clearer.

Combining Sets

So far, you have seen only two ways to represent set values: set variables and set constructors. Such set values can also be combined to form expressions with set results. The operators are +, −, and *. These symbols are also used with numeric operands to perform arithmetic. They have different meanings when the operands are set values, but they have the same precedence whether they function as set operators or arithmetic operators.

The + operator forms a set union. If A and B are set values with members of the same type, then the value of A + B (or B + A) is the set that contains all members of A and all members of B.

For examples of the use of set unions, suppose that you have the declarations:

```
VAR
 Caps, Lowers, Letters, Digits, AlphaNumerics:  SET OF char;
```

and the assignments

```
Caps := ['A'..'Z'];
Lowers := ['a'..'z'];
Digits := ['0'..'9'];
```

Then we can conveniently make the following assignments for Letters and AlphaNumerics:

```
Letters := Caps + Lowers;
AlphaNumerics := Letters + Digits;
```

A common use of the union operator is to add a single new member to a set. Suppose that you wish to add the dollar character ($) to the set AlphaNumerics:

```
AlphaNumerics := AlphaNumerics + ['$']
```

The — operator forms a set difference. If A and B are set values with members of the same type then the value of A — B is the set that contains all members of A that are not members of B. For example, the value of the expression

```
[chr(0)..chr(255)] - Letters
```

is the set of all character values that are not letters. You can also use the difference operator to remove a single value from a set. For example, the value of the expression

```
Letters - ['A']
```

is the set of all letters except the letter A.

The * operator forms a set intersection. If A and B are set values with members of the same type, then the value of A * B (or B * A) is the set that contains all members of A that are also members of B. For example, suppose that you have the declarations

```
VAR
 Commands, Options:  SET OF char;
```

and the assignments

```
Commands := ['A', 'S', 'M', 'D', 'E', 'O'];
Options := ['B', 'O', 'D', 'H']
```

then the value of the expression

```
Commands * Options
```

is the set containing the characters O and D.

Comparing Sets

Two sets that have the same base type can be compared using the = and <> operators, to see if they are equal or unequal. They can also be compared using the <= and >= operators, to see if one set contains the other. These operators produce boolean results.

The same symbols are used for arithmetic comparison; they have different meanings when the operands are set values, but they have the same precedence whether they function as set comparisons or arithmetic comparisons. The comparisons > and < cannot be used with sets.

With set operands, the = operator denotes set equality and the <> operator denotes set inequality. Two sets are said to be equal if they contain exactly the same elements, and unequal otherwise.

With set operands, the >= operator means "includes." Set A includes set B if every member of B is also a member of A. Note that A may contain other members that are not in B. Similarly, the <= operator means "is included in."

Strings

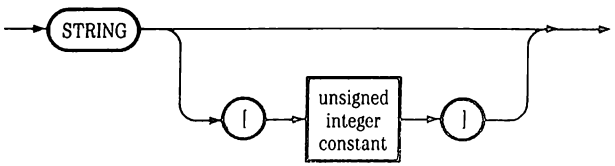
A string value is a sequence of up to 255 characters. Strings are supported by a set of predeclared functions, described later on in this chapter.

Chapter 2 contains information on string constants.

A **string variable** is a variable whose value at any point during program execution is a string. String variables are usually treated as single units, but it is possible to pick out a single character value from a string variable by indexing it in the same way that an array element is selected by indexing.

A string variable is created by declaring it with a string type. The syntax of a string type is shown in Figure 7-4.

Figure 7-4. String Type Syntax



Here are some examples of string variable declarations:

```
VAR
 MessageBuffer: STRING[200];
 InputName, OutputName: STRING;
```

The number in brackets, if used, specifies the maximum **size** of the string. The number can be any integer from 1 through 255. If no number is specified the maximum size is 255. Because the string variable's value can change during execution, the system keeps track of the **length** of the string value; if this length exceeds the maximum, an error occurs.

Important

Although it's legal to use the default string size of 255, use of memory is more efficient when a maximum size is supplied (as long as the specified size is less than 255). Whenever possible, determine the largest value for a string and declare the string with that value as the size attribute.

The value of a string variable can be altered by using an assignment statement with a string constant or another string variable:

```
Title := '    This is a title    '
```

or

```
Title := Name
```

or by means of the *readln* procedure as described in Chapter 10:

```
readln(Title)
```

or by means of the predeclared string functions, described later in this chapter.

A string value can be compared to any other string value, regardless of length. Also, as previously mentioned, a string value can be compared to a one-dimensional packed array of characters if the length of the string is the same as the number of elements in the array.

The relational operators =, <>, <, >, <=, and >= are used. One string is "less than" another if it would come first in an "alphabetic" list of strings based on the ordering of the ASCII character set. In the ASCII character set, all uppercase letters come before all lowercase letters. This can lead to unexpected results when you compare strings. For example:

See Appendix F for the ASCII character set.

```
Apple < Boy
Boy < apple
Zoo < apple
```

Elements of a String

The individual characters within a string are indexed from 1 (not 0!) to the length of the string. For example, if Title is the name of a string and we have the assignment

```
Title := 'A Quick Brown Fox'
```

then Title[1] is a reference to the first character of Title, namely the character "A", and Title[17] is a reference to the last character, namely "x". The index must not be less than 1 or greater than the length of the string. For example, Title[18] would lead to an error.

Important

While a one-character string constant is the same thing as a character constant, a string variable whose value is a single character is not the same thing as a character variable; however, you can assign either one to the other.

Predefined String Functions

In the following descriptions, a "string value" means a string variable, a string constant, or any function or expression whose value is a string. Unless otherwise stated, all parameters are value parameters.

The Concat Function

The *concat* function returns a string value. The *concat* function can take any practical number of actual parameters, each of which is a string value; the parameters are separated by commas. The syntax of the call is:

concat(String1, String2 [,String3 .. StringN]);

The *concat* function returns a string that is the concatenation of all the strings passed to it. For example, if ManualName is a string variable, then the statement

```
ManualName := concat('Apple II ', ManualName)
```

has the effect of appending the value of ManualName to the string constant "Apple II ." Another example: if you have the assignments

```
FirstName := 'Blaise';
LastName := 'Pascal'
      .
      .
      .
BothNames := concat(LastName,', ',FirstName)
```

then the statement

```
writeln(BothNames)
```

will print

```
Pascal, Blaise
```

The Copy Function

The *copy* function returns a string value. The *copy* function takes three parameters:

copy (String, Index, Count)

where String is a STRING value parameter and both Index and Count are integer value parameters. The *copy* function returns a string containing the number of characters specified by Count, copied from String, starting at the "Indexth" position in String. An example:

```
TL := 'KEEP SOMETHING HERE';
Kept := copy(TL, pos('S', TL), 9);
writeln(Kept)
```

This will print:

```
SOMETHING
```

If Count is less than or equal to zero, a null string is returned. If Index is less than one, or if Index + Count is greater than the length of String, an error does not occur. However, only the characters that lie within the specified range will be copied.

For example,

```
NewString:= 'May 25, 1978';
copy(NewString,1,20);
```

returns all of the characters in NewString, without an error, even though the string is only 12 characters long.

The Include Function

The *include* function modifies the value of a string variable. The *include* function takes three parameters:

include (Substring, String, Index)

where Substring is a string value parameter, String is a string variable parameter, and Index is an integer value parameter. The *include* function inserts Substring into String at the Indexth position in String. An example:

```
Date := 'April 1985';
Day := '1, ';
writeln(include(Day, Date, pos('1', Date)));
```

This will print:

```
April 1, 1985
```

You can insert a substring at the end of a string by using index value *length*(String)+1. For example

```
S1 := 'ABC'
S2 := 'DE'
S := include(S2,S1,length(S1)+1);
```

will produce a string of length 5 containing

```
'ABCDE'
```

The Length Function

The *length* function returns the length of a string. The *length* function takes one parameter:

length(String)

where String is a STRING value parameter.

For example, if you have the assignment

```
S := 'abcdefg';
```

then the value of *length*(S) is 7 and the value of

```
S[length(S)]
```

is "g".

The Omit Function

The *omit* function deletes a substring of a specified length from a specified position within a string value and returns the result. This is the syntax for the *omit* function:

omit (String, Index, Count)

where String is a string variable parameter and both Index and Count are integer value parameters. The *omit* function removes Count characters from String, starting at the Index specified. Example:

```
Spanish := 'Que linda es la playa blanca.';
writeln(omit(Spanish, pos('blanca', Spanish), length('blanca')));
```

This will print:

```
Que linda es la playa.
```

The Pos Function

The *pos* function returns an integer value. The *pos* function takes two parameters:

pos (Substring, String)

where both Substring and String are STRING value parameters. The *pos* function scans the string to find the first occurrence of the substring within the string. The *pos* function returns the index within the string of the first character in the matched pattern. If the pattern is not found, *pos* returns zero.

For example, suppose that you have a STRING variable named NewStudent. Then the value of *pos*('Johnson', NewStudent) will be 0 if the string does not contain the substring "Johnson." If the value of NewStudent is "Johnson, Greta" the value of the expression will be 1. If the value of NewStudent is "Greta Johnson" the value returned will be 7.

Important

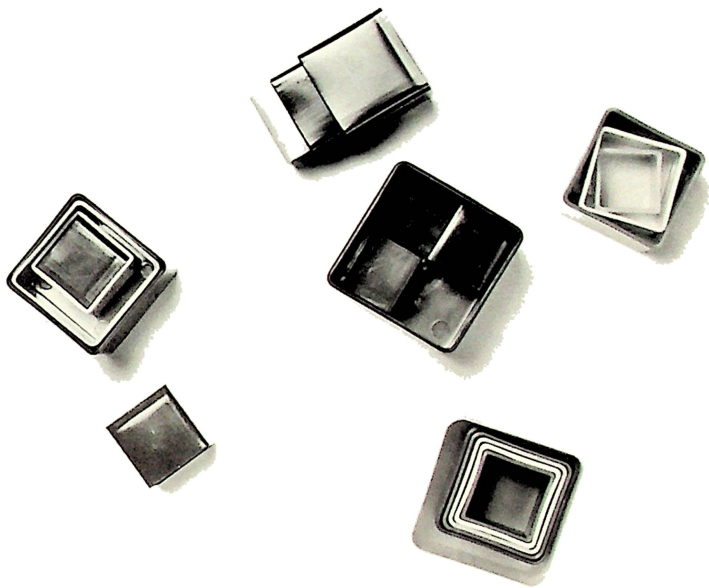
The empty string is a subset of every string. For example,

```
E:= ''    {empty string}
F:= ''    {another empty string}
N:= ('Not an empty string');
```

In this case, the value of "pos(E,N)" is 1, and the value of "pos(E,F)" is 0. Although logically the empty string could be viewed as a subset of a second empty string, *pos* returns 0, so that the value of a string can be tested for the empty string, thus avoiding runtime errors.

Chapter 8

Records



A **record** is a collection of components called **fields**, which may be of different types. Each field has its own identifier within the record, and can be individually referenced; or the record can be referenced as a whole.

As will be seen in later chapters, record types are extremely useful in conjunction with dynamic variable allocation and files; in this chapter, the discussion is restricted to ordinary record variables (which are neither dynamic variables nor components of files).

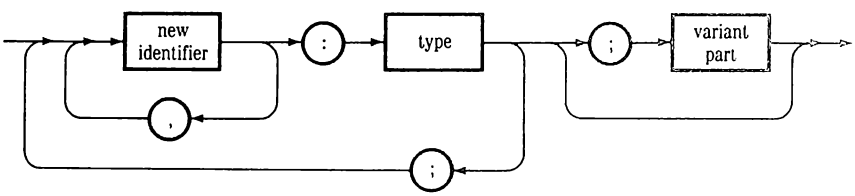
Defining a Record

The syntax for a record type is shown in Figure 8-1.

Figure 8-1. Record Type Syntax

The syntax for a field list is shown in Figure 8-2.

Figure 8-2. Record Field List Syntax



Each field identifier is the name of a distinct component, or field, within that record type. The type of a field may be anything except a FILE type. The syntax for the variant part is given in the next section; in this section, the examples assume that there is no variant part.

The following record type can be used to represent a date:

```
Date = RECORD
 Day, Year:  integer;
 Month:  STRING[3]
END;
```

If you declare a variable named Today, of type Date, then the fields of this variable can be referenced as Today.Day (an *integer* variable), Today.Year (another *integer* variable), and Today.Month (a STRING variable).

The record type in the following example uses the type Date for two of its fields. It could be used in a checking-account program.

```
TYPE
 Check = RECORD
 CheckNumber:  integer;
 DateWritten, DatePaid:  Date;
 Amount:  real;
 Recipient:  STRING[20];
 Bounced:  boolean
END;
```

Chapter 10 contains an example of how to perform input and output using a file with components of type Check.

Type Check contains six fields. A check book might be represented as an array of records of type Check:

```
VAR
 Checkbook: ARRAY[1..100] OF Check;
```

To reference a field in a record, merely write a reference to the record, then a period, then the field identifier. For example, with the declarations above,

- Checkbook[3] refers to a particular check (a variable of type Check).
- Checkbook[3].CheckNumber refers to the number of that check (an integer variable).
- Checkbook[3].DateWritten refers to the date on which that check was written (a variable of type Date).
- Checkbook[3].DateWritten.Month refers to the month within the date (a STRING variable).

To assign numbers to the checks in the array you could use the statement:

```
FOR I := 1 TO 100 DO
 Checkbook[I].CheckNumber := I;
```

Variant Records

The optional variant part of a record type contains one or more alternate field lists. For example, a variant part of a record could be declared to contain either a STRING or two *real* values, an *integer*, and a STRING.

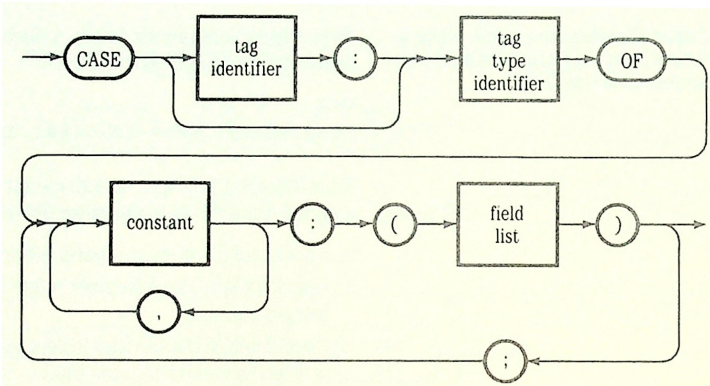
To understand what this means, recall that a Pascal variable consists of two things:

- Allocated memory space to hold the value of the variable in binary form
- Type information which tells the system how to interpret the binary information.

When a variant part is declared, the interpreter allocates enough space for the largest of the alternate field lists in the variant part. All of the alternate field lists then use the same allocated space.

The syntax for a variant part is shown in Figure 8-3.

Figure 8-3. Variant Record Syntax



The variant part resembles a CASE statement, and its meaning is closely related. The **tag identifier** and **tag type** serve to declare a **tag field**, which can be of any ordinal type. The tag field is an ordinary field of the record (just as if it were declared before the variant part).

Important

Note that the tag identifier is optional; if it is omitted, then there is no tag field. A record without a tag identifier for the variant part is called a **free-union variant**; these are described later in this chapter.

The tag type cannot be omitted, because it also relates to the constants within the variant part: each constant in the variant part must be one of the possible values of the tag identifier. Each of these constants is a label for a field list, enclosed in parentheses.

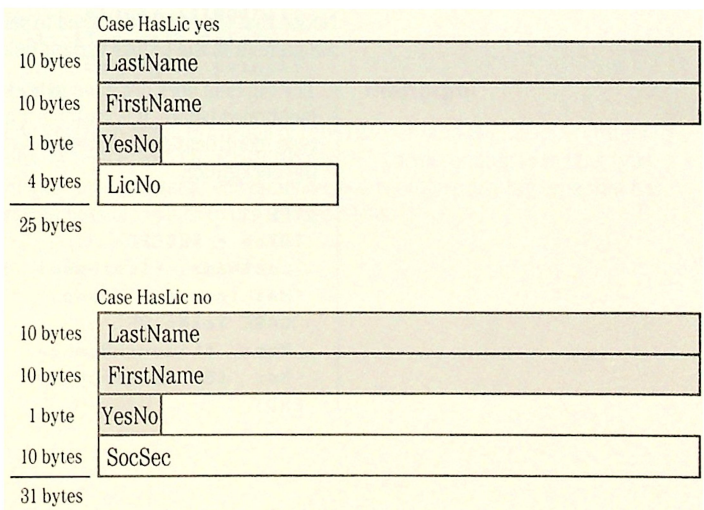
The variant part syntax allows you to declare records of a single type which can have more than one configuration—each of the field lists in the variant part represents a specific, distinct configuration.

At run time, the program can refer to any of the fields in the variant part. All of the "cases" or field lists of a variant part occupy the same space in memory, on the assumption that at any point in the program, the data in that space is to be interpreted according to just one field list.

Figure 8-4 illustrates a variant record.

```
TYPE
 YesNo = (Yes, No);
 Identification = RECORD
   LastName, FirstName: STRING[10];
   CASE HasLicense: YesNo OF
    Yes: (LicenseNumber: longint);
    No: (SocialSecurity: STRING[10])
   END;
```

Figure 8-4. Variant Records



A record of type Identification contains four fields: LastName, FirstName, HasLicense, and either LicenseNumber or SocialSecurity. The program can use the tag field HasLicense to determine which variant field, LicenseNumber or SocialSecurity, should be used.

Fields in the variant part of a record are referenced exactly as normal fields are. However, when you refer to Identification.LicenseNumber or Identification.SocialSecurity, Instant Pascal checks the tag field to see if you are referring to the active variant. If the tag is wrong for the variant you access, an error occurs.

Now suppose you have an array of records of type Identification, and some other variables to contain information input by the user:

```
VAR IdCard: ARRAY [1..100] OF Identification;
    LastNameIn, FirstNameIn, SSNumberIn: STRING[10];
    HasLicenseIn: YesNo;
    LicenseNumberIn: longint;
```

The following assignments can be made to a particular record, IdCard[N]:

```
IdCard[N].LastName := LastNameIn;
IdCard[N].FirstName := FirstNameIn;
IdCard[N].HasLicense := HasLicenseIn;
CASE IdCard[N].HasLicense OF
   Yes: IdCard[N].LicenseNumber := LicenseNumberIn;
   No:  IdCard[N].SocialSecurity := SSNumberIn
END
```

Notice that the tag field is used to select the statement that makes the assignment to the proper variant field.

Important

The tag field does not have to be declared after the word CASE; as mentioned before, it is really just another field of the record. In fact, the type Identification declared above can be thought of as an abbreviation for the declaration

```
TYPE
 IDToo = RECORD
  LastName, FirstName:  STRING[10];
  HasLicense:  YesNo;
  CASE YesNo OF
  Yes:  (LicenseNumber:  longint);
  No:  (SocialSecurity:  STRING[10])
 END;
```

where the tag field is declared before the variant part. However, note that the tag type must appear after the word CASE, since it determines the possible values for the constants in the variant part.

Free-Union Variants

In an ordinary variant record, a tag field value is stored as part of the record, and is normally used by the program to determine how to interpret the variant data. This is useful when the data in each variant field of a particular record is of a specific type; the presence of the tag field is a safeguard against misinterpreting the variant data.

It is also possible to omit the tag field from a variant record; this allows the same data to be interpreted in more than one way. A variant without a tag field is called a **free-union variant**.

A free-union variant looks like an ordinary variant, except that the tag field identifier is omitted. A tag type is still required, and case labels are still required. For example:

```
VAR
 Rect = RECORD
  CASE integer of
   0: (Top: integer;
       Left: integer;
       Bottom: integer;
       Right: integer);
   1: (TopLeft: point;
       BotRight: point);
  END;
```

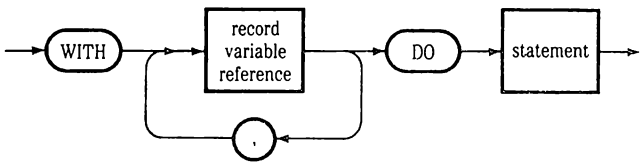
The code in this example is actually used to define rectangular shapes drawn using the predeclared graphics procedures and functions, described in Appendix C.

Now Rect.Top refers to a value of type *integer*, and Rect.TopLeft refers to a value of type Point. The labels "0" and "1", corresponding to the tag type *integer*, are a matter of convenience; you could use any tag type that has enough possible values to use as case labels.

The WITH Statement

The WITH statement is a shorthand method for referencing elements of a record. It provides a means by which the fields of specified records can be referenced using only their field identifiers. The syntax of a WITH statement is shown in Figure 8-5.

Figure 8-5. WITH Statement Syntax



The meaning of the record variable reference is determined once, before the statement following DO is executed.

Earlier, you saw the following assignments to the record IDCard[N]:

```
IDCard[N].LastName := LastNameIn;
IDCard[N].FirstName := FirstNameIn;
IDCard[N].HasLicense := HasLicenseIn;
CASE IDCard[N].HasLicense OF
 Yes:
  IDCard[N].LicenseNumber := LicenseNumberIn;
 No:
  IDCard[N].SocialSecurity := SSNumberIn
END
```

These statements can be abbreviated by using a WITH statement:

```
WITH IDCard[N] DO
 BEGIN
  LastName := LastNameIn;
  FirstName := FirstNameIn;
  HasLicense := HasLicenseIn;
  CASE HasLicense OF
   Yes:
    LicenseNumber := LicenseNumberIn;
   No:
    SocialSecurity := SSNumberIn
  END
 END
```

Variables that are not fields of records may be referenced as usual within the WITH statement. For example, if Counter is declared as a variable of type *integer*, then you can write

```
WITH CheckBook[I] DO
BEGIN
 Counter := Counter+1;
          .
          .
          .
 CheckNumber := Counter mod 100
END
```

The identifier Counter references the integer variable Counter (rather than a record field named Counter) because the records listed do not have a field named Counter.

If one of the fields of a record is itself a record, then nested WITH statements can be used:

```
WITH CheckBook[I] DO WITH DateWritten DO
 BEGIN
  CheckNumber := I;
  Day := 5;            {references CheckBook[I].DateWritten.DAY}
  Month := 'JULY'
 END
```

For convenience, these nested WITH statements can be combined into a single WITH statement:

```
WITH CheckBook[I], DateWritten DO
 BEGIN
  CheckNumber := I;
  Day := 5;            {references CheckBook[I].DateWritten.Day}
  Month := 'JULY'
 END
```

When the same field name occurs in more than one record, and both records are abbreviated via the WITH statement, a potential ambiguity arises. Consider the following example.

```
WITH CheckBook[I], DateWritten, DatePaid DO
 BEGIN
   .
   .
   .
  Day := 5;
   .
   .
   .
  END
```

The field Day occurs both in CheckBook[I].DatePaid.Day and in CheckBook[I].DateWritten.Day; which one is referenced in the assignment statement? The answer is that CheckBook[I].DatePaid.DAY is referenced, because DatePaid is the last record listed in the WITH statement.

The confusion arises because DateWritten and DatePaid are "parallel;" that is, one is not a field of the other. WITH statements that allow this kind of confusion should be avoided. For example, the following WITH construction can be used to reference fields of the same name in both DateWritten and DatePaid.

```
WITH CheckBook[I] DO
 BEGIN
  WITH DateWritten DO
   BEGIN
     .
     .
     .
    Day := 5;
     .
     .
     .
   END;
  WITH DatePaid DO
   BEGIN
     .
     .
     .
    Day := 6;
     .
     .
     .
   END
 END
```

Comparisons and Assignments

Although most assignments to records are done on a field-by-field basis, it is also possible to make assignments between entire records of the same type. For example, the result of the assignment

```
CheckBook[I] := CheckBook[I + 1]
```

is to assign to every field in CheckBook[I] the value of its corresponding field in CheckBook[I + 1].

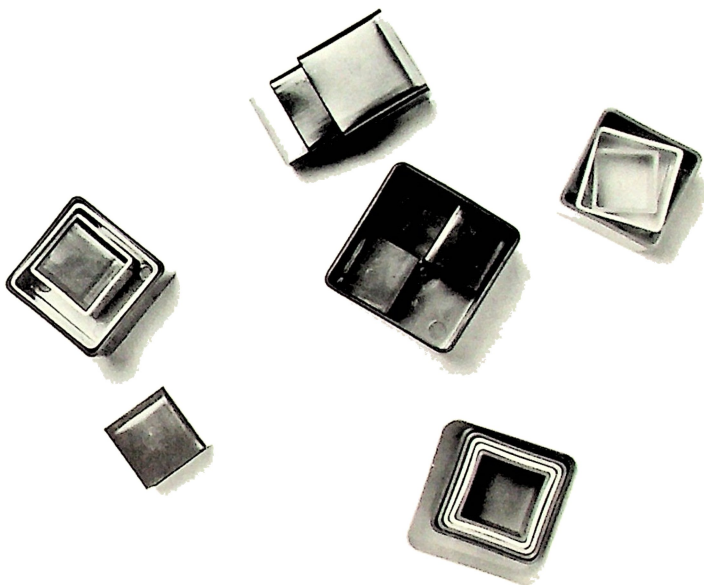
The only operations that can be performed on records are comparisons, giving boolean results. Of the relational operators, only

=	equal to
<>	not equal to

can be used with records. The rules for comparing two records are the same as the rules for assignment to records: the two records being compared must have corresponding fields of identical types.

Chapter 9

Pointers and Dynamic Variables



This chapter explains the use of the Pascal *pointer* type. It begins with a discussion of what pointers are and how dynamic variables are used in IP programs.

Concepts

Up to this point, you've seen only static variables: these are declared in the program, and the interpreter responds to the declaration by

- allocating memory space to hold the value of the declared variable at run time;
- associating the declared identifier with that memory space, and with the type information in the declaration.

In other words, when a static variable is declared, a set amount of memory is allocated for use by that variable. For example, if you declare an array variable with 100 elements of type *integer*, memory is allocated for each of the 100 elements, even if in the course of executing the program you only need to use 25 out of the 100.

Some programs need a more flexible kind of variable. For example, imagine a program that will read a sequence of records from a file or from the terminal; suppose that the records are of type

```
TYPE
 RType = RECORD
  IntVals:  integer;
  RVal:  real;
 END;
```

At the time when the program is written, the difficulty is that you don't know how many records will be read. How can you create variables to hold the records? One way is to make a generous guess and declare an array. For example if you think the number of records will not exceed 100, you could declare

```
VAR
 Rec: ARRAY [1..100] OF RType;
```

This is not very satisfactory; the program won't be able to handle a situation requiring more than 100 records, and if the number is less than 100, you will have wasted space. The situation calls for the use of **dynamic variables**.

A dynamic variable is not declared, has no identifier, and is created as needed when the program executes. Because it has no identifier, it can only be referenced indirectly, by means of a **pointer**. In Pascal, pointers and dynamic variables are two parts of a single mechanism. Note that this is the only Pascal mechanism for accessing the unallocated memory space that is available at run time.

Instead of declaring the Rec array as shown in the previous examples, you declare

```
VAR
 Ptr:  ^RType;
```

This creates a static variable called Ptr, which is a pointer to a dynamic variable of type RType. In the program we will use the pointer to allocate memory space with the call

```
new(Ptr)
```

The *new* procedure is a Pascal predefined procedure. The call shown does two things when the program is executed:

- It creates a new variable of type RType, somewhere in unused memory space.
- It sets the value of Ptr so that it points to this new variable.

Now we can refer to the new dynamic variable by writing

```
Ptr^
```

which means "the thing pointed to by Ptr." We can refer to the fields within the variable by writing Ptr ^ .IntVals or Ptr ^ .RVal, just as if "Ptr ^ " were the identifier of the variable. However, another *new*(Ptr) call will create a second variable of type RType, and then Ptr will point to this second variable. At this point you may be wondering how to refer to the first dynamic variable after the second is created, and this point will be dealt with at length further on.

Pointer Values

Pointer values are generated by *new* and there is no way to write a pointer value explicitly. Therefore you cannot declare a pointer constant; however, there is a predefined pointer constant named NIL, which can be a value of any pointer variable and which points to nothing.

The value of a pointer variable can be changed in only two ways: by giving the pointer variable as a parameter to *new* and by assigning it the value of another pointer variable or NIL.

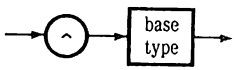
The only other operation allowed with pointers is comparison using the = and <> operators, yielding a boolean result. Pointers cannot be compared using the <, <=, >, and >= operators, and they cannot be combined using any of the arithmetic operators. Pointers are not an ordinal type, and thus do not have successors or predecessors.

For readers who are familiar with pointers in some other language, it must be emphasized that in Pascal you cannot set a pointer to a static variable; pointers can point only to dynamic variables created by the *new* procedure.

Declaring Pointer Variables

A pointer variable is declared by using a pointer type. The syntax for a pointer type is shown in Figure 9-1.

Figure 9-1. Pointer Type Declaration Syntax



In other words, a pointer type is a **domain type** prefixed with the caret (^) symbol. The domain type specifies the type of variable that a pointer of this type can point to.

Important

The domain type can be any type except a file type, and there is a special provision: the domain type can be an identifier for a type that has not yet been declared, as long as it is declared somewhere in the same type declaration as the pointer type. This is the only time in Pascal where you can use an identifier that has not yet been declared.

Examples:

```
TYPE
 DataSet = ARRAY [0..25] OF real;
 RPtr = ^RType;
 RType = RECORD
  Link:  RPtr;
  Data:  DataSet
 END;

VAR
 InData: DataSet;
 First, Last, Current: RPtr;
```

The pointer type Rptr is declared using RType as its type, even though RType is still undefined. Then RType is declared, and it makes use of the RPtr type in its own declaration, for reasons that will be explained below. The RPtr type is used again in a static variable declaration; First and Current are pointers to point to dynamic variables of type RType.

Using Pointers

The dynamic variable currently pointed to by a pointer is called the object of the pointer. To refer to the object of a pointer, merely refer to the pointer and append the ^ symbol.

For example, suppose that you have the declarations shown above, and the program sets up some values in the array InData (perhaps by reading from a file). Now you write

```
New(First);
```

which creates a dynamic variable of type RType and sets the pointer First to point to it. The data from the InData array can be transferred into the new dynamic variable by writing

```
First^.Data := InData
First^.Link := NIL
```

The second statement initializes the Link field, using NIL as a value.

Next suppose that the program puts another set of values into InData, and you want to transfer these into a second dynamic record. If you again use *new*(First), you will lose the pointer to the first dynamic record; so instead you write

```
Last := First;
{The first record is also the last record}
```

You will leave the value of First unchanged for the rest of the program; it will always point to the first dynamic record. Now, for each new dynamic record required by the program, you use the following:

```
{Create new dynamic record, pointed to}
{by last record's Link field:}
new(Last^.Link);
{Make the new record the last record}
Last := Last^.Link;
{Transfer data into last record}
Last^.Data := InData;
{Set last record's Link field to NIL}
Last^.Link := NIL
```

The first statement creates a new dynamic record and sets the Link field of the previously created record to point to the new record. The second statement sets the Last pointer to point to the new record, and the third transfers the new data into the new record. The last statement sets the Link field of the previously created record to NIL, so that the program can always tell which dynamic record is the last in the list.

When all the data has been stored in dynamic records, the result is a **linked list** of data arrays:

- The pointer First points to the first record.
- The Link field of each record except the last points to the next record in the list.
- The last record can be identified by the fact that its Link field is NIL.

Suppose that Process is a procedure that takes an ARRAY [0..25] OF *real* as its parameter, and you want to use Process on each of the data arrays in the dynamic records. You write

```
{Set current pointer to first record:}
Current := First;
{Process data of current record while}
{Current points to something:}
WHILE Current <> NIL DO
BEGIN {Advance to next record:}
 Process(Current^.Data)
 Current := Current^.Link
END
```

A general discussion of linked data structures is beyond the scope of this manual, but the above example shows the essential idea: each record in a linked data structure contains at least one pointer, which can point to another record in the structure. At least one static pointer variable (First in the example) provides an entry into the structure; after using this pointer to begin with, other records are accessed by using the pointers contained in the records.

Important

When a program starts running, pointer variables are not initialized; they have unknown values until they are set by *new* or by assignment. A variable reference using a pointer that has not been set is a reference to some unknown memory location and can have disastrous results. All pointer variables should be initialized to NIL to avoid this problem.

The New Procedure

As you've already seen, *new* is a predefined procedure that takes a pointer variable as a parameter. The *new* procedure creates a dynamic variable, whose type is the same as the pointer's domain type, and sets the value of the pointer variable to point to the new dynamic variable. The *new* procedure is the only way to create a new pointer value, and the only way to create a dynamic variable.

This is the syntax for the *new* procedure:

new(pointer identifier);

The Dispose Procedure

The *dispose* procedure destroys the dynamic variable pointed at by the value of the pointer variable. This is the syntax for the *dispose* procedure:

dispose(pointer identifier);

The pointer identifier must refer to a pointer variable that was previously assigned by *new* or that was assigned a meaningful value by an assignment statement. An error occurs if you attempt to dispose of a pointer variable whose value is undefined or is NIL. Disposing of a pointer whose dynamic variable is pointed to by another pointer can lead to undetected errors in the program if the other pointer is subsequently used to access the (now nonexistent) dynamic variable.

Chapter 10

Introduction to Files and I/O

This chapter introduces the basic concept of Pascal files, including the declaration of files in a program. It also includes detailed information on each of the predefined procedures and functions for performing **input** and **output**, or **I/O**.

This chapter also includes a description of ProDOS® pathnames and the two predefined procedures for returning the current pathname and altering the current pathname.

At the end of the chapter, there is a section with examples that illustrate how files and I/O are used in IP programs.

An Introduction to Files

A Pascal file is a special kind of structured variable. A file variable resembles an array, in that it consists of a sequence of distinct variable components, all of the same type. However, the number of components is unknown, and they are not accessed by indexing but by using the predefined I/O procedures and functions.

External Files

Although file variables can be used in the same ways as other Pascal structured variables, they are usually used to store values outside of memory. One of the most common uses is to store data files on disk. A file on a disk is one example of an **external file**. An external file is a peripheral device, or a named disk file. In order for a program to read or write information using an external file, a file variable must be declared and then associated with the external file.

Important

A file variable that is used without being associated with an external file is called an **anonymous file**. Although file variables are almost always used only to store information in an external file, there are times when you might want to use a file in the same way you'd use another type of structured variable. In that case, you would use an anonymous file. The term anonymous file is also used to refer to a file that is used for temporary storage on an external device during the execution of a program, but is destroyed when the program terminates.

Important

Any reference to an external file must be in the form of a ProDOS pathname or to the name or slot associated with a non-file device. A section at the end of this chapter explains pathnames briefly, including discussion of partial pathnames and prefixes.

Instant Pascal includes predefined functions for determining the value of the current prefix and for changing the current prefix. These are also described at the end of this chapter.

The syntax for using files to access non-file devices (such as printers and modems) is also explained at the end of the chapter.

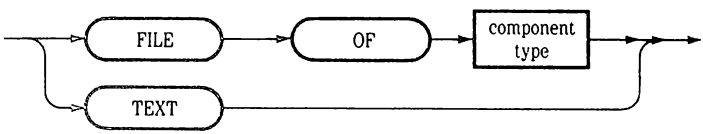
File Variables

The most important feature of a file variable is that its components are not generally in memory, but you can access them as though they were. The components exist outside the program as the contents of an external file.

A file variable is declared along with other variables, using a file type. The type of a file is determined by the type of its components.

The syntax for file type declarations is shown in Figure 10-1.

Figure 10-1. File Declaration Syntax



Important

A file may have components of any type that is not a file type, or a structured type that contains a file type component at any level of structuring. For example, the following type declaration would be illegal.

```
TYPE
 Grades = ARRAY[1..10] OF FILE;
 School = FILE OF Grades;
```

The components of a file variable are called logical records, or simply "records," although their type is not necessarily a record type. For example, the declarations

```
VAR
 IntVals:  FILE OF integer;
 RealVals:  FILE OF real;
 CompVals:  FILE OF RECORD
  I:  integer;
  R:  real;
END;
```

create three file variables.

- IntVals is a file variable whose logical records are *integer* values.
- RealVals is a file variable whose logical records are *real* values.
- CompVals is a file variable whose logical records are actually of a record type, with an *integer* value and a *real* value in each record.

Important

External disk files also have "types," which are independent of Pascal file variable types. Instant Pascal uses two types for external files: text files and data files. The structure of these external files is determined by ProDOS. Text files store ASCII characters; data files store data in the form of their internal representation.

For example, integer values stored in a text file are stored as sequences of digits, each one represented by its ASCII code. The same values stored in a datafile are represented in the binary form of the value.

Text files are created and accessed using the predefined file type. A data file results when a file variable of any other type is associated with an external file.

The performance of the predefined procedures and functions for I/O described later in this chapter varies depending upon the type of the file variable. The type of the external file doesn't affect these procedures.

The Predefined File Type Text

The Instant Pascal predefined file type *text* can be used to store any type of data, in character format.

For example, this declaration creates a file variable of type *text*:

```
TYPE
 NameFile = text;
```

Remember that a file of type *text* can be accessed only one line or one character at a time. There are many cases where this isn't the most efficient way to perform I/O.

For example, if you want to store floating-point values in a file, using a file of type *text* would mean converting each value to its equivalent in ASCII characters before it is stored in the file. Each time a value is read from the file, it would have to be converted back to its binary representation.

This is a case where you would want to declare a special-purpose file of a type other than *text* or FILE OF *char*. Another case would be any time you are using record variables. A record can be read or written only one field at a time when using a file of type *text*. If you declare a file of the record type you're using, you can read and write one entire record at a time.

Keep these guidelines in mind when reading the descriptions of the predefined procedures and how they differ when used with files of type *text* or files of other component types.

The Predefined Files Input and Output

Instant Pascal sees all peripheral devices, such as the keyboard and the Text Window portion of the screen, as external files.

Pascal communicates with the keyboard and the Text Window by using the predefined files *input* and *output*. The *input* file is a predefined file variable associated with the keyboard. The *output* file is a predefined file associated with the display device—specifically with the Text Window. Both of these predeclared files are of type *text*.

When you put a *write* statement in an IP program, *output* is the default destination. You send information to other external files by supplying an additional parameter to *write* and *writeln* statements.

For example, the statement

```
write('Hello');
```

would display "Hello" in the Text Window. If you declared a file of type *text* named "T" and associated it with the file /NewVol/Names, this statement

```
write(T,'Hello');
```

would write the word "Hello" to /NewVol/Names.

Both *input* and *output* can be associated with external files other than the keyboard and Text Window. For example, the statements

```
close(output);
reset(output, '/NewVol/Names');
```

would end the association between *output* and the Text Window and reopen *output* for use with /NewVol/Names. Then, the statement

```
write('Hello');
```

would not write the word "Hello" in the Text window; it would write it directly to /NewVol/Names.

Important

The files *input* and *output* are automatically opened for use without being declared. Procedures and functions that can be used with files of type *text* can use them, directly (receiving them as parameters), or indirectly (as the default when the file variable parameter is omitted), without declaring them first.

However, the ANSI Standard specifies that *input* and *output* must appear in the program heading if they are used. Many versions of Pascal rely on this. If you are writing code that may be transported to other versions of Pascal, include *input* and *output* in your program heading.

Important

If *input* is used within the program, the *input* file is opened automatically as a read-only file (as though a *reset* were performed for it) when program execution begins.

If *output* is used within the program, the *output* file is opened automatically as a write-only file (as though a *rewrite* were performed for it) when program execution begins.

Several of the predefined procedures and functions for use with files of type *text*, described later in this chapter, need not have a file variable explicitly given as a parameter. In these cases, *input* and *output* will be assumed by default, depending on whether the procedure or function is input oriented or output oriented.

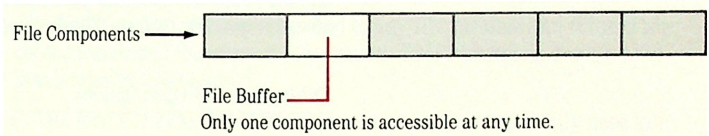
The File Buffer

Although a file variable may have any number of logical records, only one is accessible at any one time. Each logical record has a number that is its position in the file relative to the first record in the file. The first record is record zero. The position of the current record in the file is called the current file position. Program access to the current record is via a special variable associated with the file, called a **file buffer**.

The file buffer is implicitly declared when the file variable is declared. The file buffer is a variable of the same type as the components of the file.

The file buffer associated with a file variable is referred to by using the file variable's identifier, followed by the ^ (caret) symbol. For example, the current file position of a file called Names could be accessed using the file buffer Names^. This is illustrated in Figure 10-2.

Figure 10-2. The File Buffer



At any time, there is only one component of a file that may be accessed directly through the file buffer. For example, the current file position of Names is the component number of the component currently accessible through Names^. Whenever a file is opened, using any of the procedures described in this chapter, the current file position is set to component zero—the beginning of the file.

Important

Under certain conditions, such as when the current file position is at the end of the file, the value of f^ is said to be undefined. It is an error to attempt to use the value of f^ when the value is undefined. Assignment to f^ is, however, still possible.

Opening a File

For a file variable to be used it must be opened. Three procedures are provided for opening existing files and creating and opening new files. These are *reset*, *rewrite*, and *open*. Each of these procedures is explained in detail later in this chapter.

A new file may be created and opened

- By using *rewrite*, which creates a new file for write-only, sequential access;
- By using *open*, which creates a new file for read/write, sequential, or random access.

An existing file may be opened

- By using *reset*, which opens the existing file for sequential, read-only, access;
- By using *open*, which opens the existing file for read/write, sequential, or random access;
- By using *rewrite*, which opens the existing file for sequential, write-only, access.

Each of these procedures sets the current file position to zero. You can also use the *reset* and *rewrite* procedures to set the current file position of an already-open file to zero. These rules are summarized in Table 10-1.

Table 10-1. File Open Options

This Procedure:	Performed on a File of This Type:	Has This Effect:
reset	existing	opens for read-only/sequential access
reset	new	an error occurs
open	new	creates a new file for read/write/random access
open	existing	opens an existing file for read/write/random access
rewrite	new	creates a new file for write-only/sequential access
rewrite	existing	opens an existing file for read/write, sequential access; contents erased

Closing a File

If you want to redirect input or output by associating a file variable with a different external file, you must first close the open file, using the *close* procedure. This procedure is described in the section "Predefined Procedures and Functions for All Files."

Sequential Versus Random Access

Files are normally accessed sequentially. That is, when a file is opened, the first logical record is read or written, and then the current file position moves to the numerically next logical record in the file.

However, files opened with *open* can be accessed randomly with the *seek* procedure. The *seek* procedure takes a parameter that is a number referring to the sequence of logical records. By using the *seek* procedure, you can jump from one record to another, in any order.

The function *filepos* may be applied to any file variable and returns the record number of the current file position. This function is described in detail later in this chapter.

Important

The terms "random file" and "sequential file" are commonly used but misleading. Random and sequential are two methods for accessing files—not two kinds of files. Any file can be accessed randomly or sequentially, or both ways. The predefined procedures that support random access are generally used with nontext files, but are not restricted to use with nontext files. Whether it makes sense to use a particular kind of access depends on the program and on the contents of the file's logical records.

Notation

The notation used to explain the predefined I/O procedures and functions in the remainder of this chapter is a little different from that used in other parts of this book. Here's an example of the format:

Syntax: *write*(f, e_1 [, e_2, ..., e_n])

This represents the syntax of the actual parameter list of the predefined procedure *write* (when used on a nontext file).

- The terms f, e_1, e_2, and e_n stand for actual parameters. Notes on the types and interpretations of the parameters are given for each procedure or function.
- The notation e_1, e_2, ..., e_n means that any number of actual parameters can appear here, separated by commas.
- Square brackets [ ] indicate parts of the syntax that can be omitted.

The example shows that *write*, when used with a nontext file, must be passed parameters that correspond to f and e_1. Additional parameters are optional.

The procedures described in this section can be applied both to files of type *text* or FILES OF *char*, as well as to files that contain other types of logical records.

The Reset Procedure

Rewind: To move the file buffer back to the first component of the file. The term originated when files were stored on magnetic tapes, which were literally rewound when a *reset* was issued.

The *reset* procedure opens an existing file for sequential, read-only access or **rewinds** an open file.

Syntax: *reset*(f [, title])

The term "f" refers to a file variable. The term "title" is an optional expression with a string value. If a title is given, the file must not be open. If a title is not given, the file must be open.

The string should be a valid ProDOS pathname (for example, '/STUDENTS/GRADES'), or a device name for a nondisk-oriented device (such as 'MODEM:').

Important

The term "title," when used to refer to a file on a disk-oriented device, means the ProDOS pathname of the file. More information on pathnames can be found at the end of this chapter, in the *Pocket Guide to Instant Pascal*, or in the *ProDOS User's Guide*, or the *ProDOS Reference Manual*.

The *reset* procedure can be used with the predefined file *input* to read information from a file on a disk or on another device into an Instant Pascal program. For example,

```
close(input);
reset(input, '/Students/FileName')
```

This statement would associate the predefined file variable *input* with an existing file called FileName on a volume called Students. The file buffer would be set to the first component of the file, which could then be read sequentially.

The statement *reset*(f), when the file specified by f is already open, causes the file to be rewound. If the file was originally opened with the *rewrite* procedure, it becomes read-only.

Notice that *reset* preserves the contents of an existing file, unlike *rewrite*, which erases the current contents of any file on which it is used.

An error occurs if there is no existing external file with the name specified by "title."

The following conditions always hold after *reset*(f [, title]) is executed:

- *eof*(f) is *true* if the file is empty. Otherwise, *eof*(f) is *false*.
- The current file position is the first component of the file (component zero), and the file buffer variable f ^ contains the value of that component unless *eof*(f) is *true*, in which case the value of f ^ is undefined.

The Rewrite Procedure

The *rewrite* procedure creates and opens a new, empty file for sequential, write-only access, or rewinds and erases an open file.

Syntax: *rewrite*(f [, title])

- The term "f" is a file variable. If a title is given, the file must not be already open, or an error occurs.
- The term "title" is an optional expression with a string value. The string should be a valid name for a file on a file-oriented device (a ProDOS pathname), or a device name (for example, 'MODEM:').

rewrite(f) (with no title specified) when f is not yet open creates an empty anonymous file which can be written to.

rewrite (f) when the file specified by f is already open causes the file to be "rewound," that is, the current file position is reset to the beginning of the file, and any prior contents of the file are deleted. If the file was originally opened with the *reset* procedure the file becomes write-only.

rewrite(f, title) creates a new external file with the name "title," and associates the file variable "f" with this external file. If an external file with the name "title" already exists, it is deleted and a new empty file with the same name is created in its place.

The following conditions always hold after *rewrite*(f [, title]) is executed:

- *eof*(f) is *true* (either because the file is new, or because the contents have just been erased).
- The current file position is component zero; that is, the first component written to the file will become the first component of the file. The value of f ^ is undefined, and remains undefined until something is written to the file.

The Open Procedure

The *open* procedure opens an existing file or creates and opens a new file for random, read/write access.

Syntax: *open*(f, title)

- The term "f" refers to a file variable. The file must not already be open.
- The term "title" is an expression with a string value. The string should be a valid filename, or a device name (for example, 'MODEM:').

Important

You cannot use the *open* procedure with the devices 'KEYBOARD:' and 'TEXTWINDOW:'. See the section on "Input and Output With Non-File Devices" for more information.

open(f, title) opens an existing external file with the name title, and associates the file variable specified by f with this external file. If an external file with the name title does not already exist, a new empty file is created. The file is opened for both reading and writing.

The following conditions always hold after *open*(f, title) is executed:

- *eof*(f) is *true* if the file is empty. Otherwise *eof*(f) is *false*.
- The current file position is component zero, and the file buffer variable f^ contains the value of that component (unless *eof*(f) is *true*).

The Eof Function

The *eof* function detects the end of a file and returns a *boolean* value.

Syntax: *eof*[(f)]

The term "f" refers to a file variable. If f is omitted, the function is applied to the predefined file *input*. The file must be open, or an error occurs.

The *eof* function returns *true* in the following cases:

- The value is *true* if the file position is beyond the last component of the file.
- The value is *true* if the file contains no components.
- After a *get*, *eof*(f) returns *true* if the previous file position was the last component of the file.
- After a *put*, *eof*(f) returns *true* if the component written by the *put* is now the last file component.

In all other cases, *eof* returns *false*.

See the descriptions of *get* and *put* later in this section for more information on their use.

The Close Procedure

The *close* procedure closes an open file. It ends the association between the file variable and the external file, if one exists.

Syntax: *close*(f)

- The term "f" refers to a file variable. "f" must be open and must not be an anonymous file.

close(f) closes "f"; that is, the association between f and its external file is broken, and the file system marks the external file "closed." All subsequent references to f are invalid (except to open it again). In particular, the value of f becomes undefined.

A file is closed automatically in the following cases:

- If a procedure or function block having the file variable local to it is exited and the file is not already closed.
- If a program terminates with any file still open, the file is automatically closed.

The Get Procedure

The *get* procedure does two things:

- It advances the current file position to the next component.
- It reads the new current component into the file buffer.

Unlike *read*, described later in this section, it does not assign the contents of the file buffer to a variable.

Syntax: *get*(f)

The term "f" refers to a file variable. The file must be open, or an error occurs.

When a *get* is performed, if no next component exists, then *eof*(f) becomes *true* and the value of f^ becomes undefined.

The Put Procedure

The *put* procedure does two things:

- It writes the file buffer into the file at the current file position.
- It advances the file position to the next component.

Syntax: *put*(f)

The term "f" refers to a file variable. The file must be open, and the value of f^ must not be undefined (*eof* must not be *true*).

put(f) writes the value of f^ to the external file at the current file position and advances the current file position to the next file component. If the new file position is beyond the end of the file, *eof*(f) becomes *true*, and the value of f^ becomes undefined.

If *eof*(f) is *true*, *put*(f) effectively appends the value of f^ to the end of f and *eof*(f) remains *true*.

The Seek Procedure

The *seek* procedure allows you to access any component in a file. It can be used only on a file that has been opened with *open*.

The *seek* procedure does two things:

- It sets the current file position to component *n*.
- It reads the new current component into the file buffer.

Syntax: *seek*(f, n)

The term "f" is a file variable. The file must have been opened with the *open* procedure.

The term "n" is an expression with a *longint* value that specifies a file component number in the file. Components in files are numbered from zero.

The value of f^ becomes the value of that component unless n is greater than the number of the last component of the file, in which case *eof*(f) becomes *true* and the value of f^ is undefined. Thus, *seek*(f, *maxlongint*) always sets the current file position to the end of file.

For example,

seek(Names, 18)

causes the file buffer variable associated with file Names to point to the nineteenth component of the file.

The Filepos Function

The *filepos* function returns the component number of the current file position.

Result Type: *longint*

Syntax: *filepos*(f)

The term "f" refers to a file variable. The file must be open.

The *filepos* function returns a value of type *longint* that is the file component number of the current file position.

Using the Predefined Procedures and Functions With Nontext Files

The predefined procedures *read* and *write* can be used with either *text* files or files of other types. This section describes how they are used with nontext files.

Using the Read Procedure With a Nontext File

The *read* procedure reads a file component into a variable.

Syntax: *read*(f, v_1 [,v_2, ..., v_n])

The term "f" is a file variable. The file must be open.

Each v is a variable of a type that is assignment compatible with the component type of f. For example,

read(NewVals, SubTotal, Total);

If NewVals is a FILE OF *extended*, then both SubTotal and Total must be variables of a type that is assignment compatible with the *extended* type. If, in this example, NewVals is of type *integer*, an error occurs.

Using the Write Procedure With Nontext Files

The *write* procedure writes the value of an expression to a file component.

Syntax: *write*(f, e_1 [, e_2, ..., e_n])

The term "f" is a file variable. The file must be open.

Each "e" is an expression with a type that must be assignment compatible with the component type of "f."

Using the Predefined Procedures and Functions With Text Files

This section describes input and output using file variables of type *text*. Text files are distinguished from other kinds of files (for example, files of *char*) by the special significance given to the end-of-line character. This character allows a file of type *text* to be treated as a sequence of lines, rather than a sequence of individual characters. An entire line may be read from the file into a STRING type variable using the *readln* procedure, and an entire line may be written to the file by using the *writeln* procedure.

Important

The end-of-line character is tested for using the EOLN function, described later in this chapter. When the value of the file component at the current file position of a file f is an end-of-line character, it appears in the file buffer as a space character (ASCII 32).

The *read* and *write* procedures can be applied to any type of file. The versions described in this section will perform any necessary conversions to character data.

However, *readln* and *writeln* depend upon the presence of the *eoln* character, which only appears in files of type *text*.

Important

All of the predefined procedures and functions in this section need not have a file variable explicitly given as a parameter (in addition to EOF, as described previously). In these cases, one of the predefined files, *input* or *output*, will be assumed by default, depending on whether the procedure or function is input-oriented or output-oriented. Remember that *input* and *output* are predeclared as files of type *text*.

Reading and Writing Noncharacter Values to Text Files

The versions of *read* and *write* described in this section allow you to read and write values that are not of type *char*, translating them to and from their character representation. For example, *read*(f,i), where "f" is a *text* file and "i" is an *integer* variable, reads a sequence of digits (a digit being one of the characters "0" through "9"), interprets that sequence as an *integer* value, and stores it in "i."

The same call, *read*(f,i) where "f" is a nontext file and "i" is an *integer* variable, assumes that the file is positioned at a correct internal representation of an integer value, and would read that value directly into i without performing a conversion from the character representation.

If all the values stored in a file are numeric, and you don't want to examine the file as text in other environments, then this conversion process is unnecessary. In that case, you would not declare the file as type *text*. Instead, you would declare a file of one of the numeric types, or one of the structured types whose base type is one of the numeric types.

Using the Read Procedure With a Text File

The *read* procedure reads one or more values from a text file into one or more program variables.

Syntax: read([f,] v_1 [, v_2, ..., v_n])

If "f" is given, it must refer to a variable of type *text* or of type *char*. The file must be open. If "f" is omitted, it is assumed to be the predefined *text* file *input*.

Each "v" refers to a variable of one of the following types:

- *char* or a subrange of type char
- one of the integer-types or a subrange of one of the integer-types
- one of the real-types
- an ordinal type (including *boolean*) or a subrange of an ordinal type
- a *string* type

Read With a Char Variable

A *read* performed with a *char* type variable is considered equivalent to this compound statement:

```
BEGIN
 v := ff^;
 get(ff)
END
```

In this example, v is a variable of type *char* and ff^ is a FILE OF *char*. Remember that if the current file position is at an end-of-line character, ff^ contains a space character.

Read With an Integer-Type Variable

A *read* procedure performed on a text file, with an integer-type variable, reads a sequence of characters that form a signed, whole number in the range of type *integer* or type *longint*. If the sequence of characters is a valid representation of an integer, the integer value is assigned to the variable. Otherwise, an error occurs.

See Appendix B for more information on assignment compatibility.

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of a signed whole number, or when *eof*(f) becomes *true*.

It is an error if a signed whole number is not found after skipping any preceding blanks and end-of-line characters.

The following conditions are *true* immediately after a *read* from a text file with an *integer* variable:

- The current file position is over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- *eof*(f) is *true* if the last character in the numeric string was the last character in the file.
- *eoln*(f) is *true* if the last character in the numeric string was the last character on the line.

Read With a Real-Type Variable

A *read* performed on a text file with a real-type variable reads a sequence of characters that forms a signed number. If the sequence of characters is a valid representation of a real-type value, the value is assigned to the variable. Otherwise, an error occurs.

The binary-to-decimal and decimal-to-binary conversions that allow reading and writing of real-type values with text files are explained in Appendix E.

In reading the sequence of characters, blanks and end-of-line characters preceding the first digit or the sign are skipped. Reading ceases as soon as a character is reached that, together with the characters already read, does not form part of a signed number, or when *eof*(f) becomes *true*.

An error occurs if a valid signed number is not found after skipping any preceding blanks and end-of-line characters.

Immediately after a *read* from a text file with a real-type variable, the following conditions are *true*:

- The current file position is over the character following the last character in the numeric string, unless the last character in the string was the last character in the file.
- *eof*(f) is *true* if the last character in the numeric string was the last character in the file.
- *eoln*(f) is *true* if the last character in the numeric string was the last character on the line.

Read With a String Variable

A *read* performed on a text file with a string variable reads a sequence of characters up to, but not including, the next end-of-line character, or until the end of the file. The resulting character string is assigned to the variable. It is an error if the number of characters read exceeds the size attribute of the variable.

Important

With a string variable *read* does not skip to the next line after reading. An end-of-line character is left waiting in the file buffer; *read* expects a character in the file buffer. For this reason, you cannot use successive *read* calls to read a sequence of strings, as they will never get past the first line. After the first *read*, each subsequent *read* will see the end-of-line (instead of a character) and will read a zero-length string.

Instead, use *readln* to read string values; *readln* skips to the beginning of the next line after reading.

The following conditions are true immediately after a *read* from a text file with a string variable:

- The current file position is over the character following the last character in the string, unless the last character in the string was the last character in the file.
- *eof*(f) will return *true* if the last character in the string was the last character in the file.
- *eoln*(f) will return *true* unless *eof*(f) is *true*.

Read With an Enumerated Type Variable

A *read* performed on a text file with an enumerated type variable reads a sequence of characters that form an identifier. If the identifier read is identical (except for the case of the letters) to an enumerated constant of the type of the variable, the value of the constant is assigned to the variable. Otherwise, an error occurs.

In reading the sequence of characters, blanks and end-of-line characters preceding the first letter of the identifier are skipped. Reading stops as soon as a character is reached that, together with the characters already read, does not form part of an identifier, or as soon as *eof* is true.

An error occurs if an identifier is not found after skipping any preceding blanks and end-of-line characters.

The following things are true immediately after a *read* from a text file with an enumerated type variable:

- The current file position is over the character following the last character read, unless the last character in the string was the last character in the file.
- *eof*(f) will return *true* if the last character read was the last character in the file.
- *eoln*(f) will return *true* if the last character read is the last character on the line.

The Readln Procedure

The *readln* procedure is an extension of *read*. It reads a sequence of characters until the next character is the end-of-line character. It then skips to the beginning of the next line in the input file. Because *readln* depends upon finding the end-of-line character, it can be used only with files of type *text*.

The parameters allowed with *readln* are the same as those for *read*. In addition, you can use *readln*:

- with no input variables—*readln*(sourcefile)
- without any parameters.

If the first parameter does not specify a file, or if there aren't any parameters used, the procedure reads from the standard file *input*.

When *readln* is used without input variables, it causes the current file position to advance to the beginning of the next line (if there is one, else to the end of the file).

The following conditions are true immediately after a *readln*, regardless of the type of any input variable used:

- *eof*(f) will return *true* if the line read was the last line in the file.
- *eoln*(f) will return *false* unless the line following the line read is empty.

The Write Procedure

The *write* procedure writes one or more values to a text file.

Syntax: *write*([f,] p_1 [,p_2, ..., p_n])

In this diagram, f (if given) refers to a file variable of type *text*. The file must be open. If f is omitted, the procedure writes to the predefined file *output* (which is predeclared as type *text*, and associated with the IP Text window).

Each *p* in the diagram is a **write parameter**. Each write parameter includes an **output expression**, whose value is to be written to the file. Remember that the term "expression" includes single variables or constants.

As explained below, a write parameter may also contain the specifications of a field-width and a number of decimal places. Each output expression must have a result of type *char*, type *integer*, a real-type, type STRING, type *boolean*, or an enumerated type. At least one write parameter must be present.

Write Parameters

Each write parameter has the form

OutExpr [: MinWidth [: DecPlaces]]

where OutExpr is an output expression. MinWidth and DecPlaces are optional expressions with *integer* values. For example, in this statement:

```
write(NewVals, Total);
write(NewVals, Total:8:4);
```

"Total" is the output expression. The value of the real-type variable Total is to be written to file NewVals. In the first case, the optional MinWidth and DecPlaces specifications are omitted. In the second case, MinWidth is 8 and DecPlaces 4. Total will be written to a field eight spaces wide and with four characters to the right of the decimal place.

MinWidth specifies the minimum field width. MinWidth must be greater than zero. Exactly MinWidth characters are written (using leading spaces if necessary), except when OutExpr has a value that must be represented in more than MinWidth characters, in which case the exact number of characters needed are written (except when OutExpr is a STRING value). MinWidth can be used with an OutExpr of any type that is valid in a *write* parameter.

Fixed-point: A decimal representation of a real-type value that includes a decimal point and no exponent part.

Floating-point: A representation of a real-type value that is comprised of an exponent and a mantissa.

DecPlaces specifies the number of decimal places to be used in the fixed-point representation of a real-type value. It can be specified only if OutExpr has a real-type value, and if MinWidth is also specified. If specified, it must be greater than zero. If DecPlaces is not specified, and the value is one of the real-types, a floating-point representation is written.

Write With a Char Value

If *write* is given a variable of type *char*, and MinWidth is not specified, the character value of OutExpr is written to the specified file. If MinWidth is included, exactly MinWidth-1 spaces are written, followed by the character value of OutExpr. For example,

write(Names, Initial); *write*(Names, Initial:3);

In the first example, the character that is the value of Initial is written to the file, without leading spaces. In the second example, Initial is written, preceded by two spaces.

Write With a String Value

The results of a *write* with a string variable depend upon the length attribute of the string that appears as the OutExpr, and whether or not MinWidth is specified.

- If MinWidth is specified, and the length of the string is less than MinWidth, then the string is written preceded by a number of spaces equal to MinWidth minus the length of the string.
- If MinWidth is specified, and the length of the string is greater than MinWidth, then the first MinWidth number of characters are written.
- If MinWidth is specified, and the length of the string equals MinWidth, or if MinWidth is not specified, the entire string value is written on the file.

In this case,

write(LastName: 8);

LastName is a string variable with a size of 10. If LastName holds a value that is either 9 or 10 characters, only 8 will be written to the Text window.

Write With an Integer Value

If OutExpr is of type *integer*, its decimal representation is written on the specified file. Assume that OutDigits is a *string* value that contains the

decimal representation of the absolute value of OutExpr, with no leading zeros unless the value of OutExpr=0, in which case OutDigits contains the single character "0".

If MinWidth is used, and its value is greater than the number of digits in the decimal representation of the value to be written, leading spaces will be written to the left of the number. The number of spaces depends upon the MinWidth specification.

If the value of OutExpr is less than zero, a "—" character is written to the file, signifying a negative value.

If MinWidth is omitted from the write parameter, then it is assumed to be zero. In this case, the decimal representation of the value is written to the file, without any leading spaces.

The representation of OutExpr is written to the file as if by this algorithm:

```
BEGIN
 FOR I := 1 to MinWidth - (length(OutDigits) + 1) DO
  write(ff,' ');
 IF OutExpr < 0 THEN
  write(ff, '-')
 ELSE
  write(ff, ' ');
 write(ff, (OutDigits);
END;
```

The term "ff" represents the variable referenced by f.

For example, if the decimal representation of the value of OutExpr is 4545, and MinWidth is given as 8, the FOR statement will write three space characters to a file. Because OutExpr is not less than 0, the IF clause will not execute, and the ELSE clause will output one more space. Finally, the last *write* statement will output the decimal number 4545 to the file.

Write With a Real-Type Value

If OutExpr has a real-type value, its decimal representation is written to the specified file. This representation is specified by the presence (or absence) of DecPlaces, and its value.

Important

This section includes detailed information on how the exact format for decimal representation of a real-type value is determined. One of the best ways to see how this works is to use the Instant Window to read and write some real-type values, given different field width and decimal place specifications.

If DecPlaces is present, a fixed-point representation is written, using the following algorithm. If DecPlaces is absent, a floating-point representation is written.

Assume that IntDigits is a *string* value that contains the decimal representation of this expression:

```
trunc(abs(OutExpr))
```

IntDigits contains no leading zeroes. (Unless, of course, the value of OutExpr is zero, in which case IntDigits contains the single character "0".) This expression is the value for the left-of-the-decimal-point portion of OutExpr.

Now assume that FracDigits is a *string* value that contains the decimal representation of this expression:

```
round((abs(OutExpr) - trunc(abs(OutExpr))) * 10^DecPlaces)
```

with enough leading zeroes to make *length*(FracDigits)=DecPlaces.

Then the fixed-point representation is written to the file using this algorithm:

```
BEGIN
 IF MinWidth >= length(IntDigits)+length(FracDigits)+2 THEN
  write(ff, ' ': MinWidth-TotalDigits-3);
 IF OutExpr < 0 THEN
  write(ff, '-')
 ELSE
  IF MinWidth >= length(IntDigits)+length(FracDigits) +2 THEN
   write(ff, ' ');
 write(ff, IntDigits, '.', FracDigits)
END
```

If DecPlaces is not specified, a floating-point representation is written. If MinWidth is omitted from the write parameter, then it is assumed to be 10.

The algorithm used to write a floating-point representation works in this way. The expression *abs*(OutExpr) can be represented in floating- point notation in this form:

$m.n \times 10^e$

In this expression, m is always a digit from 1 to 9, unless the value of OutExpr is 0. Assume that IntDigit is a *string* value that contains the decimal representation of m—a single digit. Assume that FracDigits is a

string value that contains the first MinWidth-9 digits of the decimal representation of n rounded, and with trailing blanks retained and trailing zeroes added if necessary. Assume that ExpDigits is a *string* value that contains the decimal representation of *abs*(e) with enough leading zeroes to make *length*(ExpDigits)=4. Also assume that NegExp has the value *true* if e<0, and otherwise is *false*. Given these expressions, this is the algorithm for writing a floating-point representation:

```
BEGIN
 IF OutExpr < 0 THEN
  write(ff, '-')
 ELSE write(ff, ' ')
  write(ff, IntDigit, '.', FracDigits, 'E');
 IF NegExp THEN
  write(ff, '-')
 ELSE write(ff, '+');
 write(ff,ExpDigits)
END
```

Write With a Packed Array of Char

If OutExpr is a packed array of *char*, the effect is the same as writing a string whose length is the number of components in the type.

Write With an Enumerated Type Value

If the value of OutExpr is an enumerated type, the string representation of the enumerated identifier corresponding to the value is written to the file. If the length of this string representation is less than MinWidth then MidWidth-length spaces are written out before the string. In any case the entire string representation is always written, even if the length of the representation is greater than MinWidth.

The Writeln Procedure

The *writeln* procedure is an extension of *write*. Essentially it does the same thing as *write*, and then writes an end-of-line character to the output file (ending the line).

The parameters are the same as those used with *write*, with these exceptions:

- The file type parameter is not required (the effect is to write to the predefined file *output*)
- The write parameters can be entirely omitted (the effect is to write a blank line to *output*)

If the first parameter does not specify a file variable the procedure writes to the predefined file *output* (which is a file of type *text* and is associated with the IP Text Window.)

If write parameters are not specified, an end-of-line character is written to the specified file variable.

If parameters are entirely omitted, an end-of-the-line character is written to *output*.

The following things are true immediately after a *writeln* is executed, regardless of the type of write parameters used:

- *eof*(f) returns *true* if the last character written became the last character in the file. If the file is write-only, then *eof*(f) will be *true*.
- *eoln*(f) returns *false* unless the character following the last character written is an end-of-line character.

The Eoln Function

Result Type: *boolean*

Syntax: *eoln* [(f)]

The term "f" refers to a file variable of type *text*. The file must be open. If f is omitted, the function is applied to the predefined file *input*.

eoln returns *true* if the character at the current file position is an end-of-line character. An error occurs if *eoln*(f) is applied to a non-text file, if f is write-only, or if *eof*(f) is *true*.

Important

Every line in a text file is expected to be terminated by an end-of-line character. However, it is possible for a file not to end with an end-of-line character. If a file is read-only (opened with *reset*) and the last character is not an end-of-line character, then f^ becomes a space character, *eoln*(f) becomes *true*, and *eof*(f) remains false. The next attempt to read a character will then cause *eof*(f) to become *true*. This will only happen if the file is read-only.

The Page Procedure

The *page* procedure sends a formfeed character (ASCII 12) to the designated file. If the file parameter is omitted, the character is sent to the Text Window. The result is to clear the Text Window of the current display.

If the file is write-only, and if the last character in the file is not an end-of-line character, then one is inserted before the *page* is done.

Syntax: *page*[(f)]

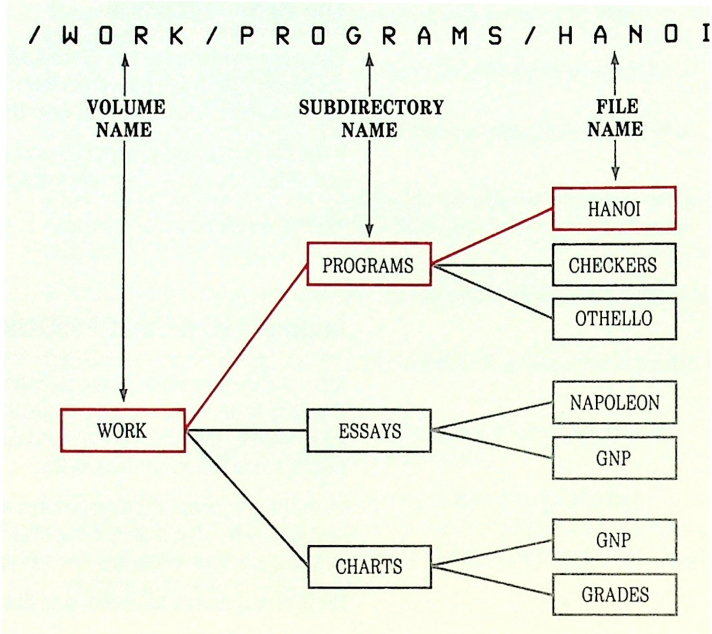
Instant Pascal and ProDOS

Instant Pascal uses the Apple II Professional Disk Operating System (ProDOS®) for handling files. ProDOS determines the organization of the disks used to store IP programs and data, and the structure of the external files that are stored on these disks.

In the descriptions of file procedures and functions throughout this chapter, any parameter that is an external file specification can be a string expression whose value is a ProDOS pathname.

ProDOS pathnames have the form illustrated in Figure 10-3.

Figure 10-3. ProDOS Pathnames



A ProDOS **pathname** is a series of filenames, each preceded by a slash (/). The first filename in a pathname is the name of a disk **directory** or the **volume name**. The last filename in a pathname is the name of a file. Any filename between the name of the volume directory and the specified file is the name of a **subdirectory**.

The disk directory is a file that holds information about the contents of the disk. Subdirectories are also files that hold information about subgroupings of files within the directory structure.

Successive file names indicate the path, from the disk directory, through subdirectories, to the file, that ProDOS must follow to find a particular file. The maximum pathname length is 64 characters.

Every procedure or function in this chapter that requires you to name an external file will also accept either a pathname or a partial pathname. A partial pathname is a portion of a pathname that doesn't begin with a slash or a volume name. The maximum length for a partial pathname is 64 characters, including slashes.

ProDOS automatically adds the current prefix to the front of partial pathnames to form full pathnames. The prefix is a pathname that indicates a directory; it is internally stored by ProDOS.

Here is an example of how an external file could be opened using the *rewrite* procedure.

```
rewrite(TestData, '/Programs/NewValues/Feb4');
```

This statement opens a new file variable, TestData, and associates it with the external file Feb4—which is is in subdirectory NewValues on the disk Programs. You could perform this same operation by first setting the prefix and then using a partial pathname:

```
setprefix('/Programs/NewValues/');
rewrite(TestData, 'Feb4');
```

More information on volumes, pathnames, and subdirectories can be found in the *Pocket Guide to Instant Pascal* and the *ProDOS User's Manual.*

The SetPrefix Procedure

The *setprefix* procedure changes the current ProDOS prefix to the value specified by a STRING parameter. The syntax is

setprefix(newprefix)

NewPrefix represents a STRING expression whose value is a ProDOS prefix, as defined above. If the volume specified by the new prefix is not on-line, or the subdirectory specified does not exist, an error occurs.

The GetPrefix Function

The *getprefix* function returns a STRING value that represents the current ProDOS prefix. The syntax is

getprefix

Input and Output With Nonfile Devices

Every peripheral device is seen by Pascal as an external file. However, instead of using a ProDOS pathname as a parameter (as you do when addressing a disk drive), you refer to them either by name, or by slot number.

A printer or modem can be referred to as PRINTER: or MODEM:, depending upon how you've setup your system. Also, the keyboard and the Text Window can be addressed as devices by using their device names, KEYBOARD: and TEXTWINDOW:.

At the start of execution, IP does an implicit

```
reset(input, 'KEYBOARD:');
rewrite(output, 'TEXTWINDOW:');
```

There are restrictions on the use of these devices. For example, you cannot perform a *reset* on PRINTER: or TEXTWINDOW: (which would make them read-only devices), and you cannot perform a *rewrite* on KEYBOARD: (making it a write-only device).

Any other device is referred to by slot number. For example,

```
reset(Infile, '1:');
```

opens the device in slot one and associates it with the file variable Infile. The slot number must appear between single quotation marks, and it must be followed by a colon.

Only one slot-oriented device may be open at a time.

Using a Nontext File to Store Record Variables

Text files can read and write only one character, or one line, at a time. There are many cases where you will want to store information in record variables, and be able to read an entire record at one time. This can be done using a non-text file.

The following record was used as an example in Chapter 8.

```
TYPE
 Check = RECORD
  CheckNumber:  integer;
  DateWritten, DatePaid:  Date;
  Amount:  real;
  Recipient:  STRING[30];
  Bounced:  boolean;
END;
```

Type Date is defined as another record:

```
TYPE
 Date = RECORD
 Day, Year:  integer;
 Month:  STRING[3];
END;
```

A program designed to keep track of checks would include procedures to perform several different tasks. For example, one of these might be to find a check made out to a particular person. Another would be to start a new file. Input and output would be done in different ways, depending upon the task.

To start a new file, a file variable must be declared and associated with an external file:

```
TYPE
 NewYear = FILE OF Check;

BEGIN
  rewrite(NewYear, '/Checkbook/Yr85');
      .
      .
      .
END
```

The type declaration creates the new file as a file of the record type Check. Each component in the file is one record.

Given a file of records of type Check, the following example shows how you would open the file and search for a check written to a particular person.

```
VAR
 Target : STRING[30];  {Target is the variable to hold the name to be}
                       {searched for}
 Temp : Check;         {Temp holds an input record while search goes on}
 Found : boolean;      {Flag to signal end of search}
    .
    .
    .
setprefix('/CheckBook');              {Sets the ProDOS prefix to volume}
                                      {/CheckBook}
reset(NewYear, 'Yr85');               {Opens file NewYear and associates}
                                      {it with external file /Yr85}
    .
    .
    .
writeln('Name of recipient:');        {Writes prompt to output}
readln(Target);                       {Reads input and assigns to Target}

Found := false;
WHILE NOT (Found OR eof(NewYear))
BEGIN
 read(NewYear, Temp);
 IF Temp.Recipient = Target THEN
 Found:=true
END;
```

This loop continues until Found is set to *true* or the end of the file is reached. The use of a nontext file allows you to look at only one field in a record, with only one operation. If you did this with a text file, you would have to read in each field of the record until you found the one you were looking for.

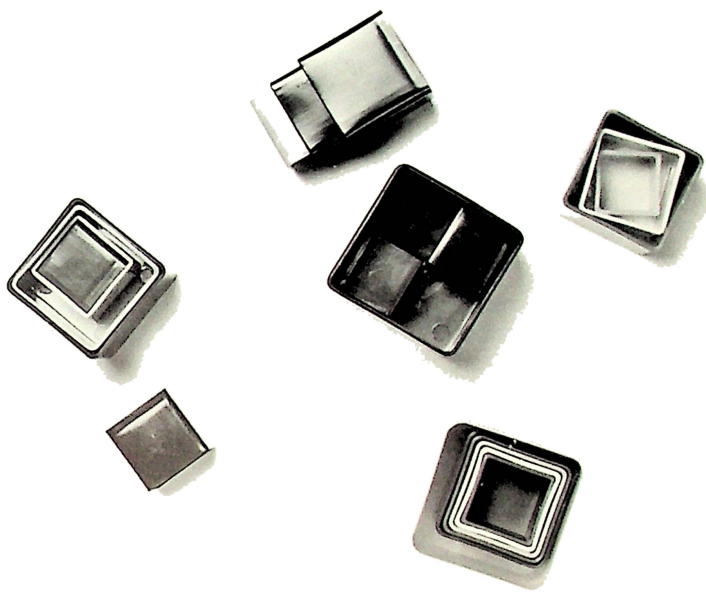
Another task might be to add a record to the end of the file. You could do this using the *open* procedure to open the file:

```
open(NewYear, 'Yr85');
seek(maxlongint);
write(NewYear, NewRec);
close(NewYear);
```

Using *maxlongint* as the parameter of the *seek* procedure automatically sets the current file position to the end of the file. This random access of the file is possible only because the file was opened with *open*.

Appendix A

Instant Pascal and Other Pascals



This appendix contains brief descriptions of the differences between Instant Pascal and Macintosh Pascal, Apple II Pascal 1.3, and ANSI Pascal.

Instant Pascal Versus Macintosh Pascal

The Instant Pascal language differs very little from Macintosh Pascal. The major differences are in the greater number of built-in procedures and functions available with Macintosh Pascal, and the ability to use the Macintosh operating system from Macintosh Pascal programs.

IP programs can be typed in exactly as written for IP and run on the Macintosh, with the following exceptions:

- Instant Pascal filenames are formatted according to the conventions for ProDOS pathnames.
- Instant Pascal predefined functions that deal with the ProDOS prefix (*setprefix, getprefix*) are not available in Macintosh Pascal.
- Instant Pascal predefined procedures and functions for using hand controllers (game paddles) with IP programs are not available in Macintosh Pascal
- Most of Instant Pascal's predefined color patterns are not available in Macintosh Pascal.
- In Macintosh Pascal, the size of the graphics pen can range from (0,0) to (32767,32767). In Instant Pascal, the size ranges from (0,0) to (255,255).

Instant Pascal Versus Apple II Pascal 1.3

In addition to Instant Pascal, Apple produces a second version of Pascal for Apple II computers. This version, Apple II Pascal, is based on UCSD Pascal. UCSD Pascal was developed in the late 1970's at the University of California at San Diego in response to the need for a good programming environment for microcomputers in colleges.

The major difference between IP and Apple Pascal is that Instant Pascal is meant to be used primarily as an educational tool. Its main purpose is to provide an easy-to-use environment for use in learning the language, rather than as the basis for large application programs.

Apple II Pascal includes extensions to the language itself for use with small systems, as well as a variety of built-in functions and procedures that aid in the development of large programs. These tools include an assembler and linker for developing assembly language subroutines for use in Apple II Pascal programs.

In addition to the ability to use assembly language, Apple II Pascal has many other built-in features that enable you to divide large programs into smaller pieces that can be easily manipulated in a machine with limited memory. These features include

- program segmentation
- user-defined libraries

Apple Pascal is also available for registered developers as a runtime system. A runtime system consists of the minimum number of pieces of the Pascal system that are necessary to boot, and then run, an application program. When you run an Instant Pascal program you must first boot the program disk, then load and run your program. You must display output, both text and graphics, in the appropriate Instant Pascal window.

An Apple Pascal runtime system allows you to make a bootable disk with your application on it. The operating environment is transparent—that is, your program controls the entire screen.

Runtime systems are available for use on 48K, 64K, and 128K Apple II computers.

Language Differences

The reserved words of both versions are the same, with these exceptions. Apple Pascal includes these additional reserved words:

- EXTERNAL
- IMPLEMENTATION
- INTERFACE
- SEGMENT
- UNIT

Instant Pascal includes the SANE data types—*single, double, comp*, and *extended*. Apple II Pascal uses only the *real* type. Also, all IP arithmetic performed on real-types uses SANE. SANE is not integrated into Apple II Pascal.

Predefined Identifiers

Instant Pascal and Apple Pascal share the following predefined identifiers.

abs	*boolean*	*char*	*chr*
close	*concat*	*copy*	*delete*
eof	*eoln*	*false*	*get*
input	*insert*	*integer*	*length*
maxint	*new*	*odd*	*ord*
output	*page*	*pos*	*pred*
put	*read*	*readln*	*real*
reset	*rewrite*	*round*	*seek*
sqr	*sqrt*	*succ*	*text*
true	*trunc*	*write*	*writeln*

Do not assume that procedures or functions with the same names work identically in both systems.

Using Instant Pascal as an Apple Pascal Development Tool

Many of the effective debugging tools included with Instant Pascal can be used to develop and debug Apple Pascal procedures. If you're careful not to include any of the special extensions of either IP or UCSD in your code, you can use IP for building individual procedures that you can later use in a large Apple II Pascal program. However, be aware that the algorithms used in certain predefined procedures and functions may differ.

Instant Pascal Versus ANSI Pascal

Instant Pascal follows the ANSI standard and includes the following extensions to that standard:

- IP includes the reserved words OTHERWISE, STRING, and USES.
- An identifier may have an underscore following the initial letter of the identifier.
- IP includes the type *longint* and the additional real-types *double*, *extended*, and *comp*.
- A signed constant identifier may denote a value of type *integer*, *longint*, or *comp*.

- In IP, the rules for mixed *integer* and *longint* arithmetic are simple: all operands are converted to *longint* before any integer arithmetic is performed, and the result is always *longint*. A *longint* value may be used wherever an *integer* value is required if the value falls in the range *−maxint..maxint*.
- Similarly, all integer-type and real-type operands are converted to *extended* before any real arithmetic is performed and the result is always *extended*. An *extended* value may be used wherever a *real*, *double*, or *comp* value is required, provided the value falls within the range of values permissible.
- IP includes the STRING type, which is compatible with *char* types.
- Individual characters within a STRING may be referenced as though the string were a one-dimensional array.
- A STRING type variable may be compared with a variable of type *char*.
- IP includes an optional OTHERWISE clause for the CASE statement.
- IP allows value and variable parameters of type STRING.
- IP supports the predefined SANE library that can be accessed with the USES clause.
- An optional second parameter may be given to *reset* and *rewrite* to associate a file variable with an external file.
- A file may be opened with *open* to allow random read/write access to a file.
- An explicit *close* procedure is supplied for those file variables associated with external files.
- The *seek* procedure can be used for random access to file components.
- The *filepos* function returns the component number of the current file position, a value that may be used in subsequent *seeks*.
- STRING and enumerated type values can be read from text files using the *read* and *readln* procedures.
- STRING and enumerated type values can be written to text files using the *write* and *writeln* procedures.
- The *ord* function always returns a *longint* result.
- IP includes a set of built-in string procedures and functions.

Appendix B

Type Compatibility

In many cases it is important to know whether two variables are **type compatible**. For example, you may want to assign the value of a variable to a second variable. Even if the variables are not of **identical types**, you may still be able to perform the assignment, if the variables are **assignment compatible**.

The following sections present the rules that define identical and compatible types. These rules apply to both the scalar types discussed in Chapter 3, and to the structured types (arrays, sets, strings, records, and files) discussed in later chapters.

There are three kinds of type compatibility:

- Types may be identical. This is required for certain types of operations.
- Types may be assignment compatible. This means that the value of one can be assigned to the other, and vice versa.
- Types may be simply compatible. Compatibility at this level is often needed before assignment compatibility can be determined.

Type Identity

Identical types are required only in the following context:

See Chapter 6 for information on variable parameters and actual and formal parameters.

- When passing a variable parameter, the actual and formal parameters must be of identical types.

Two types, T1 and T2, are identical if one of the following is true:

- T1 and T2 are the same type identifier.
- T1 is declared to be equivalent to a type identical to T2.

What the second statement implies is that T1 need not be declared directly to be equivalent to T2; thus the type declarations

```
T1 = integer;
T2 = T1;
T3 = integer;
T4 = T2;
```

result in T1, T2, T3, T4, and type *integer* all being identical types.

However, the type declarations

T5 = SET OF *char*;
T6 = SET OF *char*;

do not result in T5 and T6 being identical, since SET OF *char* is not a type identifier. (It's a set variable declaration.) Finally, note that two variables declared in the same declaration; as in

T5, T6: SET OF *char*;

are of identical type. However, if the declarations are separate (as in the first example using T5 and T6) then the above definitions apply. The declarations

v1: SET OF *char*;
v2: SET OF *char*;
v3: *integer*;
v4: *integer*;

result in v3 and v4 being of identical type (*integer* is a type identifier), but not v1 and v2.

Compatibility of Types

Compatibility between two types is sometimes required. Specific instances where type compatibility is required are noted elsewhere in this manual. Type compatibility is, more importantly, often a precondition of assignment compatibility.

Two types are compatible if any of the following are true:

- Both types are identical.
- Both types are real-types.
- Both types are integer-types.
- One type is a subrange of the other.
- Both types are subranges of identical host types.
- Both types are set types with compatible base types.
- One type is a STRING type and the other is a *char* type.

Assignment Compatibility

Assignment compatibility is required whenever a value is assigned to something, either explicitly (as in an assignment statement) or implicitly (as in passing value parameters).

A value of type T2 is assignment compatible with a type T1 (i.e., T1 := T2 is permissible) if any of the following are true.

- T1 and T2 are identical types and neither is a file type or a structured type that contains a file type component at any level of structuring.
- T1 and T2 are real-types and the value of type T2 is within the range of possible values of T1.
- T1 is a real-type and T2 is an integer-type.
- T1 and T2 are compatible set types, and all members of the value of type T2 are within the range of possible values of the base type of T1.
- T1 is a *char* type and the value of type T2 is a string type and has a length of 1.
- T1 is a STRING type with a size of n and the value of type T2 is a string type and has a length less than or equal to n.
- T1 is a STRING type and T2 is a *char* type.

It is an error if assignment compatibility is required and none of the above is true.

Appendix C

Predefined Graphics Procedures

Instant Pascal comes with a set of predefined procedures for drawing graphic images. This appendix includes an overview of how the IP graphics system works, as well as descriptions of each of the predefined graphics types and procedures.

These are the constants, types, variables, procedures, and functions that are described in this Appendix.

```
CONST
 patcopy = 8;
 pator = 9;
 patxor = 10;
 patbic = 11;
 notpatcopy = 12;
 notpatxor = 14;
 notpatbic = 15;

TYPE
 str255 = string[255];
 pattern = packed array [0..7] of 0..255;
 vhselect = (v, h);
 point = RECORD
  CASE integer OF
   0:
    (v: integer;
     h: integer);
   1:
    (vh : array[vhselect] of integer);
 END;

 rect = RECORD
  CASE integer OF
   0:
    (top: integer;
     left: integer;
     bottom: integer;
     right: integer);
   1:
    (topleft: point;
     botright:point);
 END;
```

```
penstate = RECORD
 pnloc : point;
 pnsize : point;
 pnmode : integer;
 pnpat : pattern;
END;

VAR
 Black, Magenta, Brown, Orange,
 DarkGreen, Gray1, Green, Yellow,
 DarkBlue, Purple, Gray2, Pink,
 MedBlue, LightBlue, Aqua, White: pattern;

 randseed: longint;

PROCEDURE drawline(a, b, c, d: integer);
PROCEDURE drawchar(ch : char);
PROCEDURE drawstring(s : str255);
PROCEDURE frameoval (r : rect);
PROCEDURE invertcircle(x, y, r: integer);
PROCEDURE line(dh, dv : integer);
PROCEDURE lineto(h,v : integer);
PROCEDURE move(dh, dv: integer);
PROCEDURE moveto(h, v : integer);
PROCEDURE paintcircle(x,y,r : integer);
PROCEDURE paintoval(r : rect);
PROCEDURE paintrect(r : rect);
PROCEDURE pennormal;
FUNCTION ptinrect(pt: point; r : rect) : boolean;
PROCEDURE penmode(i : integer);
PROCEDURE penpat(p : pattern);
PROCEDURE pensize(width, height: integer);
PROCEDURE writedraw(p1 [,p2,...,pn]);
PROCEDURE getpenstate( VAR p : penstate);
PROCEDURE setpenstate(p : penstate);
```

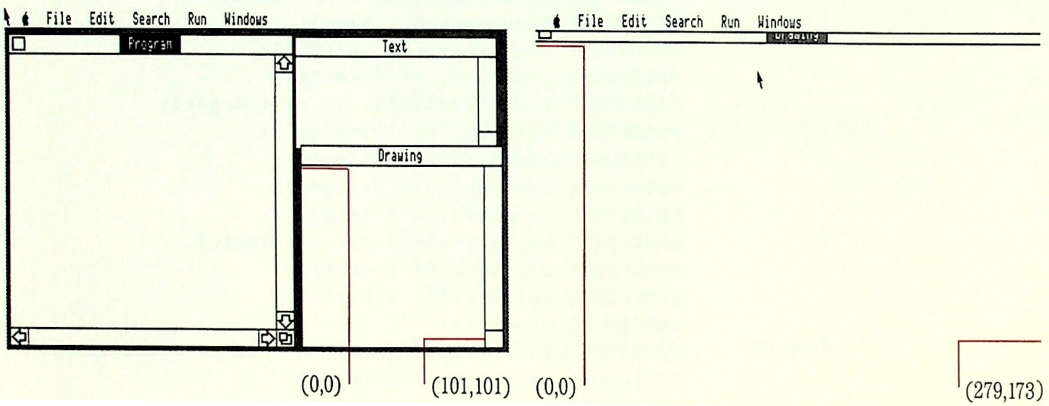
The Drawing Window

When you startup Instant Pascal, the Drawing Window represents a coordinate system 102 units wide and 102 units high. The upper left-hand corner is respresented by the coordinates 0,0. The bottom right-hand corner is represented by the coordinates 101,101.

If you grow the Drawing Window to its maximum size, it becomes 280 units wide by 174 units high. The upper left-hand corner is still 0,0—but the lower right-hand corner becomes 279, 173.

When you draw a shape using one of the graphics procedures, you specify where you want the shape in the Drawing Window by specifying points in this coordinate system.

Figure C-1. The Instant Pascal Drawing Window



See the last section of this appendix for information on using these procedures with a color display.

Each point in this system is actually a bit. If you have a monochrome display device, a bit with a value of 0 appears white. A bit with a value of 1 appears in the second color (usually black or green). When you startup IP, the Drawing Window is white. Each bit in it has a value of 0.

The Pen

The graphics pen is similar to an invisible cursor that acts like a penpoint. The pen has four characteristics:

- Location—where the pen is currently located in the Drawing Window
- Size—the dimensions of the "penpoint"
- Pattern—the design or color the pen creates
- Mode—the state that governs how the source pattern is copied to the current destination pattern

You can change each of these characteristics by using the built-in procedures and functions.

Patterns

The IP graphics equivalent of ink or paint is the **pattern**. A pattern is either a color, or a design, that's defined by an 8 bit by 8 bit rectangle. When you startup IP, the pattern being displayed in the Drawing Window is rows and columns of an 8 x 8 square. Each bit in the square has a value of one.

When you "draw" an object in the Drawing Window, you are transferring a different pattern to the screen. For example, if you paint a black rectangle (black is the standard pattern used with the procedures), you are copying the black pattern to the screen area that defines the rectangle. The black pattern is defined by an 8 x 8 square of bits, each with the value of one.

You can change the pattern you draw with by using the *penpat* procedure. The procedure takes an argument of type *pattern*. A pattern is defined as a packed array [0..7] of 0..255. You can create your own patterns by declaring a variable of type *pattern*, then assigning various values to the array.

In addition to the patterns you can create yourself, IP has several predefined pattern variables—black, white, magenta, and so on. These variables produce color on a color monitor. When you use the *penpat* procedure with one of the color variables on a monochrome system, you produce monochrome patterns in different shades of your monitor's second color.

The Transfer Mode

The simplest kind of drawing copies a pattern to the Drawing Window. The Drawing Window is the **destination** for the copy operation. The pattern you've selected is the **source** for the copy operation.

Every time you create an image, the process can be defined as the source pattern being transferred to the destination. The final result of the transfer depends upon three variables:

- The destination pattern
- The source pattern
- The transfer mode

The standard destination pattern is "white." The array that defines the pattern holds only zeros. The source pattern is "black"—the array holds only ones. The standard transfer mode is *patcopy*. This means that the source pattern is copied to the destination and the result is the source pattern, regardless of what pattern the destination currently holds.

The *patcopy* mode is one of eight transfer modes. The other seven modes transfer the source pattern to the screen by performing a *boolean* operation, using the bits in the source and destination as operands.

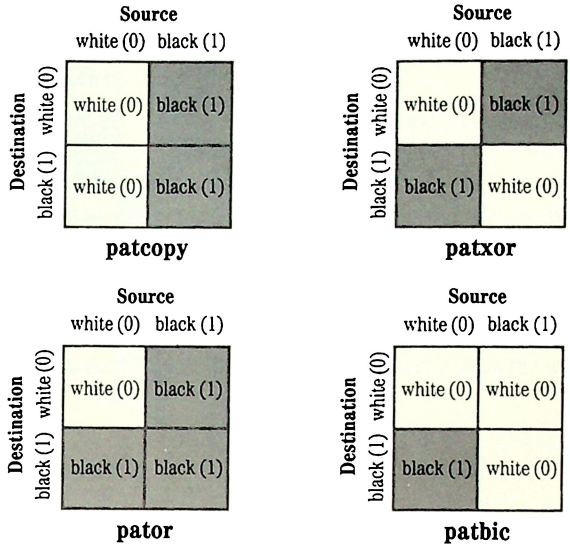
Table C-1 describes the eight transfer modes. Each mode is a constant. The integer value is used as the argument to the *penmode* procedure.

Table C-1. Transfer Modes

Transfer Mode	Integer Value
patcopy	8
pator	9
patxor	10
patbic	11
notpatcopy	12
notpator	13
notpatxor	14
notpatbic	15

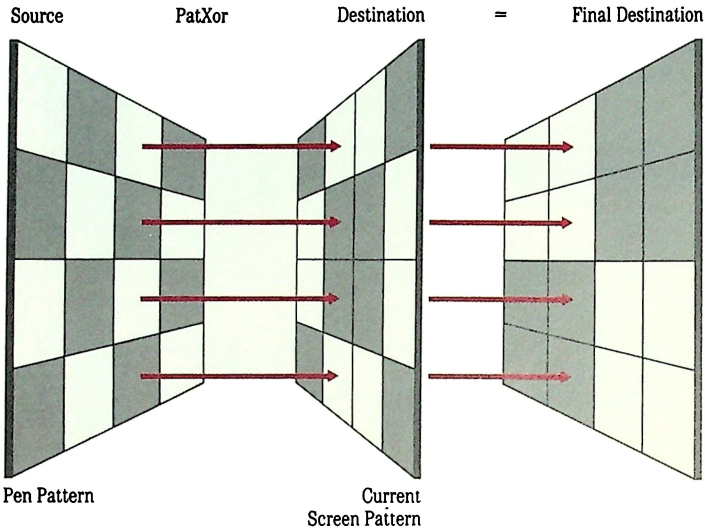
You can predict the outcome of any transfer using the tables shown in Figure C-2. Figure C-3 shows a sample pattern and the results of performing the XOR transfer operation on it.

Figure C-2. The Transfer Modes



For NOT operations *(notpatcopy, notpatxor, notpator,* and *notpatbic*), invert the source and determine the result in the same way.

Figure C-3. How the Transfer Mode Works



Using the Graphics Procedures With a Color Monitor

If you are using a color monitor, you can use one of the set of built-in color variables as the argument to the *penpat* procedure.

black	magenta	brown	orange
darkgreen	gray1	green	yellow
darkblue	purple	gray2	pink
medblue	lightblue	aqua	white

Each of these variables is of type *pattern.*

Predefined Types

There are several predefined types that are used with the IP graphics procedures. Each of these types is described below.

Every procedure that takes an argument of type *point* will also accept two integers that define a coordinate in the Drawing Window. The y coordinate value is given first, then the x coordinate.

Every procedure that takes an argument of type *rect* will also accept four integers: the first pair defines the upper-left corner of the rectangle, the second pair defines the lower-right corner of the rectangle. In each pair, the y coordinate is given first, then the x coordinate.

Procedures that don't take an argument of type *point* or type *rect*, but nevertheless require a coordinate will accept the coordinate in the usual x,y fashion.

Defining Points

A *point* is defined by this declaration of a variant record:

```
vhselect = (v,h);
point = RECORD
 CASE integer of
  0 : (v : integer;
       h : integer);
  1 :  (vh :  array[vhselect] of integer);
END;
```

The first variant, case 0, defines a point as a record with two fields: *v* (vertical), and *h* (horizontal). You could define a point using these statements:

```
VAR
 newp : point;

WITH newp DO
 BEGIN
  v := 23;
  h := 39
 END;
```

This defines a point that is 23 units from the top of the Drawing Window and 39 units from the left side.

The second variant, case 1, defines a point as an array of *vhselect*, which is a predefined enumerated type with values of *v* and *h*. Using these definitions, *vh* is an array that holds two values:

```
VAR
 newp :  point;

WITH newp DO
 BEGIN
  vh[v] := 23;
  vh[h] := 39
 END;
```

Using these data structures, you can define the coordinates of a rectangle.

Defining Rectangles

A variable of type *rect* is defined by this declaration:

```
rect = RECORD
 CASE integer of
  0 : (top : integer;
       left : integer;
       bottom : integer;
       right : integer);
  1 : (topleft : point;
       botright : point)
 END;
```

Using this predefined variant record, you can define a rectangle either by specifying four integer values (as shown below), or by first defining variables of type *point* and using these to define the rectangle.

```
VAR
 newr : rect;

BEGIN
 WITH newr DO
  BEGIN
   top := 10;
   left := 20;
   bottom := 40;
   right := 20
  END
END;
```

This code defines a rectangle using two points:

```
VAR
 newp1, newp2 :  point;
 newr :  rect;

BEGIN
 WITH newp1 DO
  BEGIN
   v := 23;
   h := 44
  END;

 WITH newp2 DO
  BEGIN
   v := 75;
   h := 90
  END;

 WITH newr DO
  BEGIN
   topleft := newp1;
   botright := newp2
  END
END;
```

Predefined Graphics Procedures and Functions

This section describes the procedures and functions you use to put text and graphic images in the Drawing Window with an Instant Pascal program.

Procedures That Write Text Into the Drawing Window

IP provides several procedures that you can use to put text in the Drawing Window. These are described in the following sections.

Each of these procedures changes the pen location to the position immediately to the right of the last character displayed.

The WriteDraw Procedure

Syntax: *writedraw*(e_1, [e_2,...e_n]);

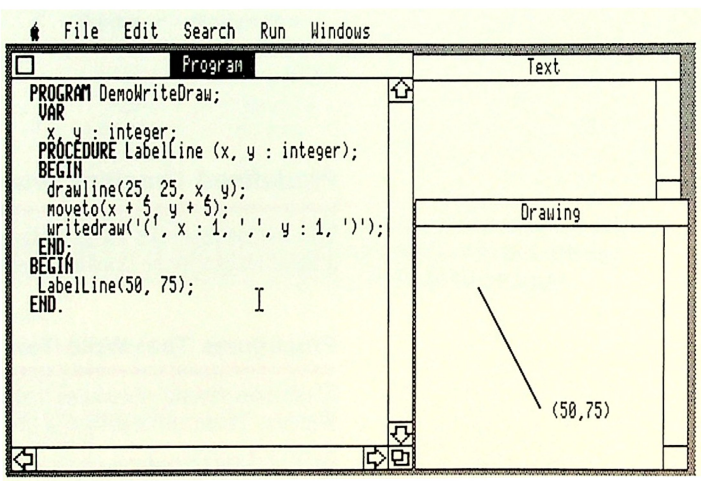
See Chapter 10 for a complete description of the *write* procedure.

The *writedraw* procedure works exactly like the *write* procedure, except that the output appears in the Drawing Window, rather than in the Text Window.

The DemoWriteDraw program, shown in Figure C-4, is an example of how the *writedraw* procedure works. In this procedure, a line is drawn from the point at coordinates 25,25 to the point specified by the arguments passed to the LabelLine procedure. In this case, the point is at 50,75.

The program then uses the *writedraw* procedure to label the point. The arguments are specified in the same way they would be if you were using the *write* command: characters or strings must appear in single quotes and the format for displaying numeric values can be specified using write parameters (as described in Chapter 10.)

Figure C-4. The WriteDraw Procedure



The DrawChar Procedure

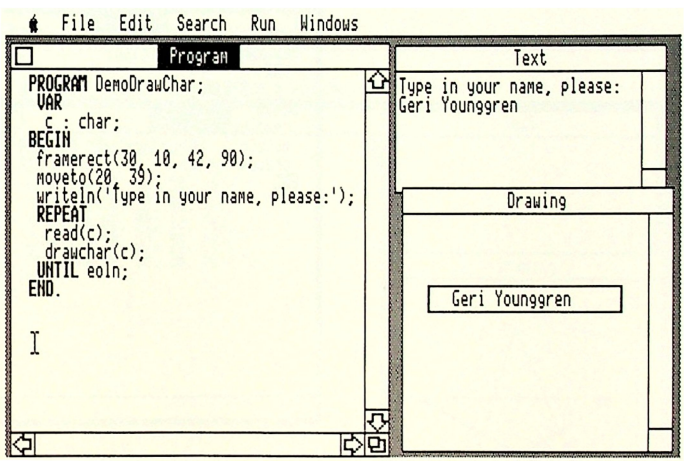
Syntax: *drawchar*(x);

The *drawchar* procedure displays a character in the Drawing Window, at the current pen location. The procedure takes an argument of type *char*.

The DemoDrawChar program, shown in Figure C-5, is an example of how the *drawchar* procedure works. The procedure reads each character entered from the keyboard. Because the Text Window is associated with the predefined file *output*, the character appears in the Text Window after it is entered. The procedure uses the *moveto* procedure (described later in this chapter) to move the pen to point 20,39. Then the *drawchar* procedure "echoes" the character to the Drawing Window.

See Chapter 10 for information on the predefined files *input* and *output*.

Figure C-5. The DrawChar Procedure



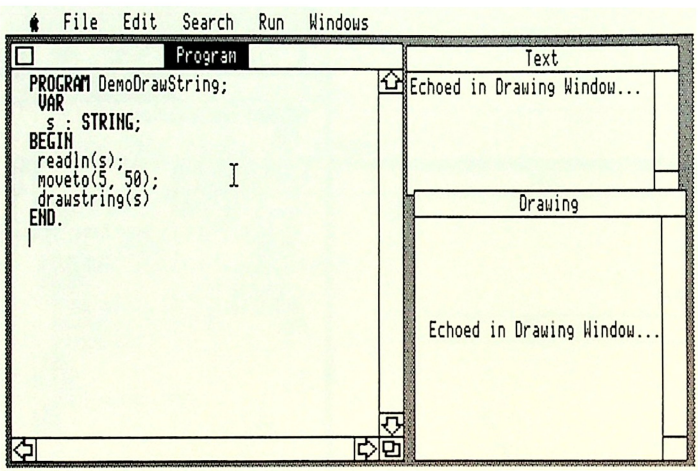
The DrawString Procedure

Syntax: *drawstring*(s)

The *drawstring* procedure takes a parameter of the predefined type *str255*, which is declared as a string with a size of 255. The procedure draws the string in the Drawing Window, beginning at the current pen position.

The DemoDrawString program, shown in Figure C-6, is almost identical to the DemoDrawChar program. It reads a string, rather than a single character, from the keyboard, and displays it in both the Text and Drawing Windows.

Figure C-6. The DrawString Procedure



Procedures That Create Shapes

You can draw lines, circles, ovals, and rectangles from an IP program by using the procedures described in this section.

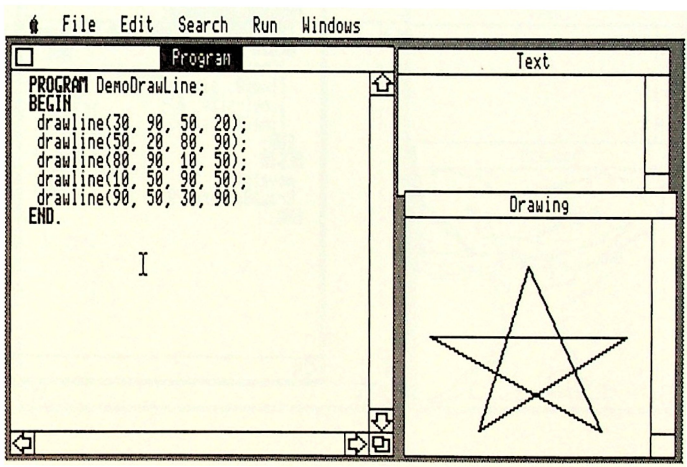
The DrawLine Procedure

Syntax: *drawline*(a, b, c, d);

The *drawline* procedure draws a line in the Drawing Window from the point represented by the coordinates (a, b) to the point represented by (c, d). Each of these parameters must be an integer value.

The DemoDrawLine program, shown in Figure C-7, presents an example of how *drawline* works. This program draws a star in the Drawing Window with five *drawline* statements.

Figure C-7. The DrawLine Procedure



The Line Procedure

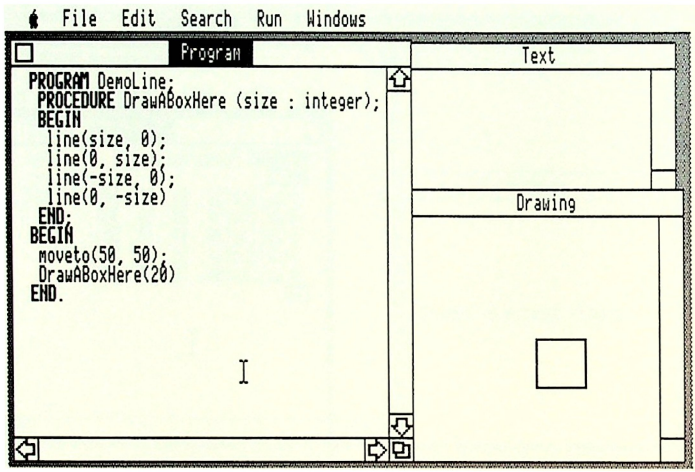
Syntax: *line*(dh, dv);

The *line* procedure draws a line from the current pen location to a point specified by the arguments. The point is calculated by adding *dh* and *dv* to the current pen location. The procedure uses the current pen location as the origin of its coordinates. For example, if the current pen location is at 50,100, this statement:

```
line(25, 25);
```

would draw a line from 50,100 to 75,125. The DemoLine program, shown in Figure C-8, is an example of how the *line* procedure can be used.

Figure C-8. The Line Procedure



The LineTo Procedure

Syntax: *lineto*(h, v);

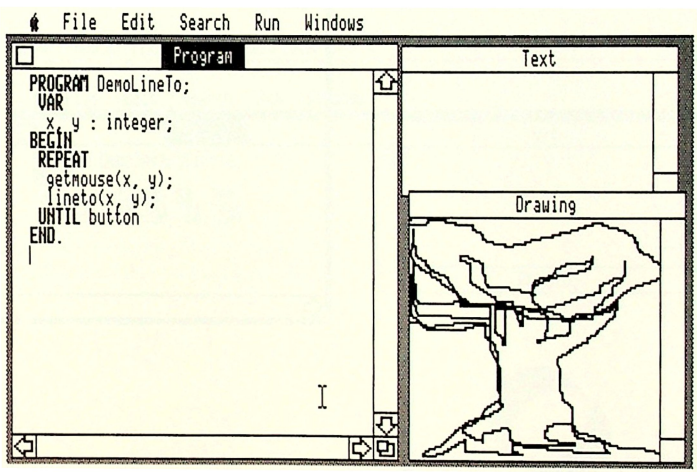
The *lineto* procedure draws a line from the current pen location to the point specified by the two arguments. Unlike the *line* procedure, the arguments refer to a point in the general coordinate system. For example, if the pen location is at coordinates 50,100, the statement:

```
lineto(25,25);
```

draws a line from 50,100 to 25,25. The DemoLineTo program, shown in Figure C-9, demonstrates how this procedure can be used with the *getmouse* procedure and the *button* function to construct a simple drawing program.

The *getmouse* procedure and the *button* function are described in Appendix D.

Figure C-9. The LineTo Procedure

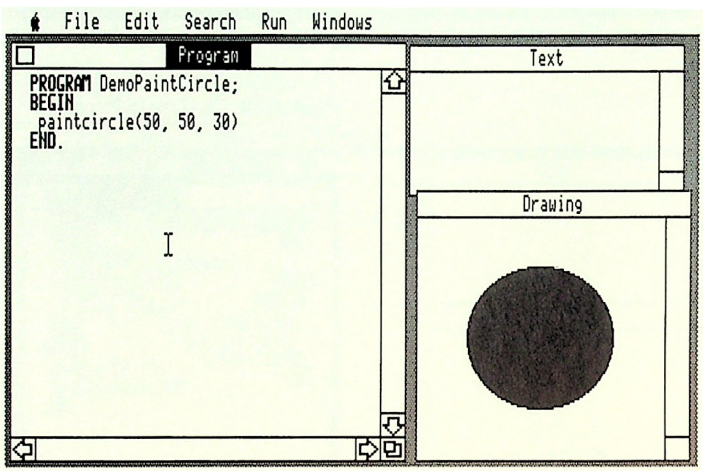


The PaintCircle Procedure

Syntax: *paintcircle*(x, y, r);

The *paintcircle* procedure draws a solid circle. The procedure takes three integers as arguments. The first two define a point that will be the center of the circle. The last argument is the radius, in units, of the circle. The DemoPaintCircle program, shown in Figure C-10, draws a circle with the center at (50,50) and a radius of 30 units.

Figure C-10. The PaintCircle Procedure



The InvertCircle Procedure

Syntax: *invertcircle*(x, y, r)

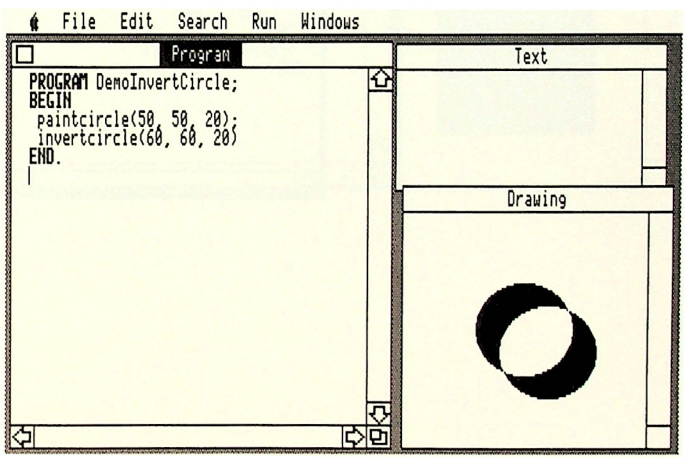
The *invertcircle* procedure reverses the value of every bit included in the circle defined by the arguments given to the procedure. For example, the statements:

paintcircle(40, 50, 50);
invertcircle(40, 50, 50);

would draw a solid circle in the Drawing Window and then "erase" it by reversing the value of each bit in the first circle.

If the first circle were drawn with a pattern other than white, the effect would be to reverse the pattern: each bit would change value. The DemoInvertCircle program, shown in Figure C-11, is an example of how the *invertcircle* procedure can be used.

Figure C-11. The InvertCircle Procedure

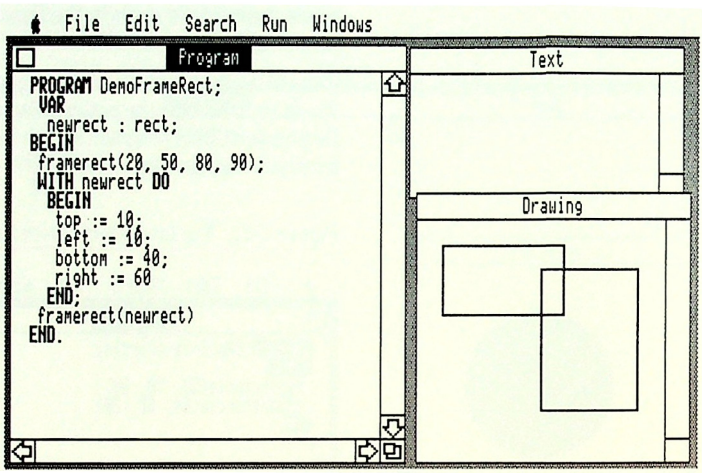


The FrameRect Procedure

Syntax: *framerect*(r)

The *framerect* procedure draws the outline of a rectangle. It takes as an argument either a variable of type *rect*, or four integers. The *framerect* procedure is illustrated in Figure C-12.

Figure C-12. The FrameRect Procedure

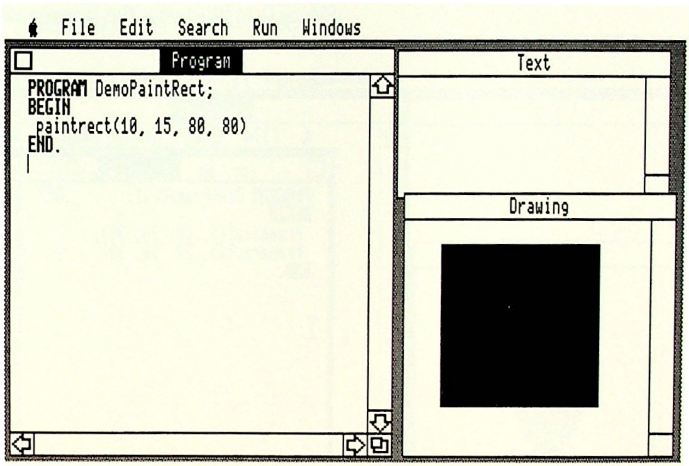


The PaintRect Procedure

Syntax: *paintrect*(r);

The *paintrect* procedure draws a solid rectangle. It takes as an argument either a variable of type *rect* or four integers. The *paintrect* procedure is illustrated in Figure C-13.

Figure C-13. The PaintRect Procedure



The FrameOval Procedure

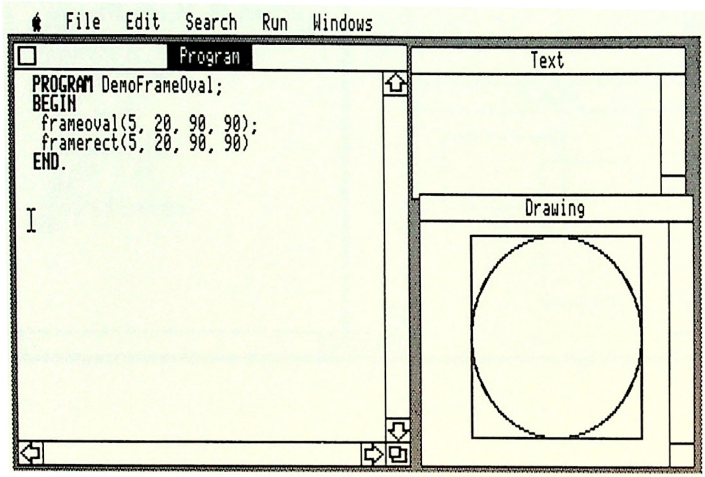
Syntax: *frameoval*(r)

The *frameoval* procedure outlines an oval which is defined by the rectangle given as a parameter. The procedure then draws the outline of an oval within the rectangle.

You can specify the rectangle in one of two ways: by passing the procedure a variable of type *rect*, or by giving it four integers.

Figure C-14 illustrates the *frameoval* procedure.

Figure C-14. The FrameOval Procedure

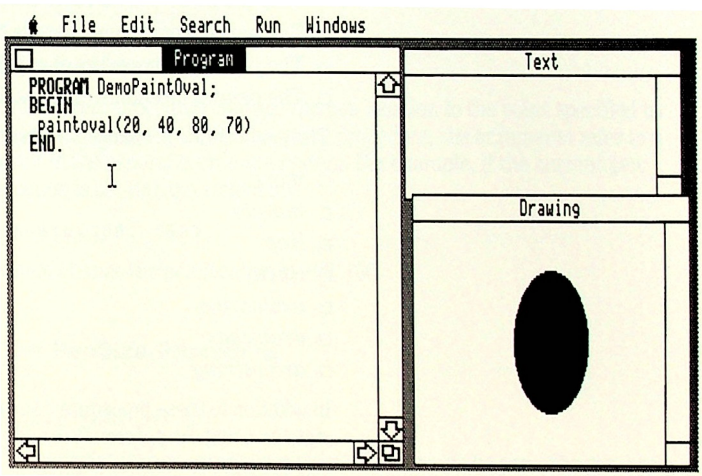


The PaintOval Procedure

Syntax: *paintoval*(r);

The *paintoval* procedure draws a solid oval inscribed within the rectangle used as an argument. As with the other procedures that take a rectangle as an argument, the rectangle can be either a variable of type *rect* or it can be specified using four integers. Figure C-15 illustrates the *paintoval* procedure.

Figure C-15. The PaintOval Procedure



Pen Procedures

There are several procedures that allow you to change the attributes of the graphics pen. These are described in the following sections.

The pen attributes are defined using a record variable:

```
penstate = RECORD
 pnloc : point;
 pnsize : point:
 pnmode : integer;
 pnpattern : integer
END;
```

- The *pensize* procedure changes the value of the *pnsize* field.
- The *penmode* procedure changes the value of the *pnmode* field.
- The *penpat* procedure changes the value of the *pnpattern* field.

The pen location is changed by these procedures:

- *move*
- *moveto*
- *line*
- *lineto*
- *writedraw*
- *drawchar*
- *drawstring*

In addition to these procedures, you can save and restore a particular pen state by using the *getpenstate* and *setpenstate* procedures.

The penstate is stored in a record of type *penstate*. You can declare a variable of type *penstate* and use it to store any combination of pen attributes.

The standard penstate is defined as:

pnsize = 1,1;
pnmode = *patcopy*;
pnpat = *black*;

This means that the pensize is one unit by one unit; the transfer mode is *patcopy*, and the pen pattern is black. This penstate can be restored at any time by calling the *pennormal* procedure.

The Move Procedure

Syntax: *move*(dh,dv);

The *move* procedure changes the location of the pen by adding the value of it arguments to the current pen location. For example, if the current pen location is 50,50 this code:

```
move(50, 50);
```

would change the location of the pen to 100, 100.

The MoveTo Procedure

Syntax: *moveto*(h, v);

The *moveto* procedure changes the pen location to the point specified by the two arguments. Unlike the *move* procedure, the arguments refer to a point in the general coordinate system. For example, if the current pen location is 50, 100, the statement:

The *moveto* procedure is illustrated in Figures C-4 and C-7.

```
moveto(100, 150);
```

would change the pen location to 100, 150.

The PenSize Procedure

Syntax: *pensize*(h, w)

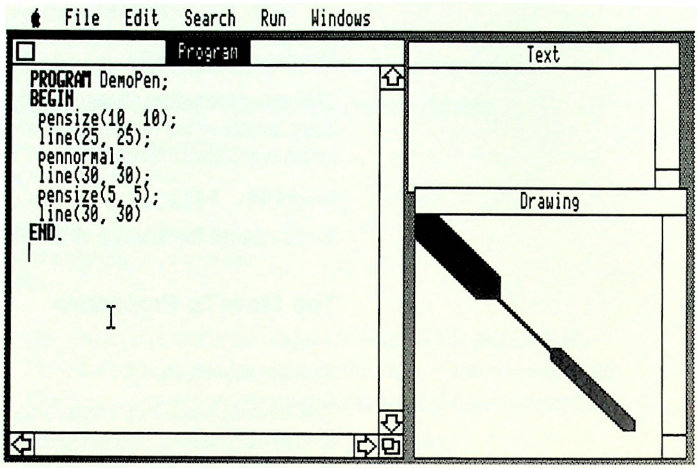
The *pensize* procedure changes the dimensions of the pen. The pen has a height and a width that are defined in the standard units used on the coordinate plane. The standard pen size is one unit high by one unit wide.

The *pensize* procedure takes two arguments; one that specifies the horizontal dimension of the pen and one that specifies the vertical dimension of the pen. The horizontal argument must be within the range 1.. 127. The vertical argument must be within the range 1..255.

The location of the pen is always specified by the upper-left corner of the current pen.

Figure C-16 illustrates the *pensize* procedure, as well as the *pennormal* procedure (which is described below).

Figure C-16. The PenSize and PenNormal Procedures



The GetPenState Procedure

Syntax: *getpenstate*(p);

The *getpenstate* procedure gets the current penstate and stores it in the variable of type *penstate* which is passed to the procedure.

The SetPenState Procedure

Syntax: *setpenstate*(p);

The *setpenstate* procedure changes the current pen attributes by replacing them with the values in a variable of type *penstate*. For example,

```
VAR
 newpen : penstate;

BEGIN
 WITH newpen DO
  BEGIN
   pnloc.h:= 10;
   pnloc.v:= 20;
   pnsize.h := 10;
   pnsize.v := 10;
   pnmode := patxor;
   pnpattern :  aqua
  END;
 setpenstate(newpen);
```

sets the current pen attributes to the values defined in *newpen.*

The PenNormal Procedure

Syntax: *pennormal*;

The *pennormal* procedure restores the standard values to the penstate.

Transfer and Pattern Procedures

The procedures defined in this section enable you to change the transfer mode and the current pen pattern.

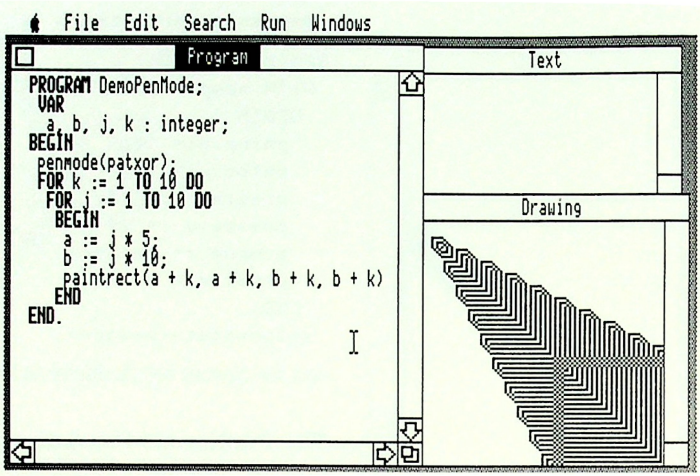
The PenMode Procedure

Syntax: *penmode*(m);

The *penmode* procedure takes as an argument an integer value that specifies one of the eight transfer modes. The eight penmodes and their integer values are shown earlier in Table C-1.

Figure C-17 illustrates the use of the *penmode* procedure.

Figure C-17. The PenMode Procedure



The PenPat Procedure

Syntax: *penpat*(p)

The *penpat* procedure sets the current pattern by taking as an argument a variable of type *pattern*. There are seventeen built-in variables that can be used with the *penpat* procedure. If you have a color monitor, you use these variables to change the colors displayed by your program. If you have a monochrome monitor, using these variables will cause your program to display various shades and patterns in the monitor's second color.

The pattern variables are shown in the preceding section on the Transfer Mode.

You can also define your own monochrome patterns by assigning values to a variable of type *pattern*. A *pattern* variable is a packed array [0..7] of 0..255.

Figures C-18 and C-19 show examples of the use of the *penpat* procedure.

Figure C-18. The PenPat Procedure

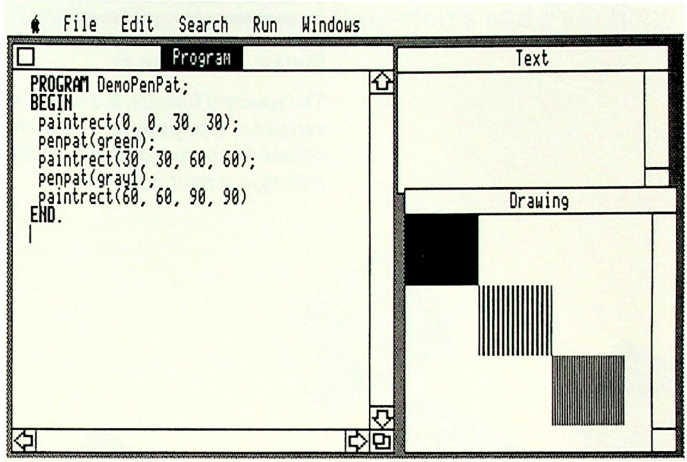
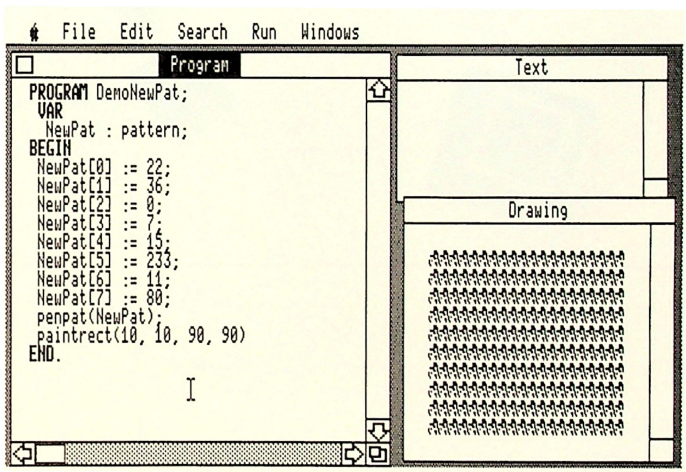


Figure C-19. Using the PenPat Procedure With a User-Defined Pattern Variable



The PtInRect Function

Syntax: *ptinrect*(pt, r);

The *ptinrect* function is a *boolean* function that takes two arguments: a variable of type *point* and a variable of type *rect*. If the *point* is in the area defined by the rectangle, the function returns *true*. If the point is not in the rectangle, it returns *false*.

Appendix D

Miscellaneous Predefined Procedures and Functions

The functions described in this appendix can be used in an IP program to detect the use of the keyboard, mouse, or game paddles.

The Button Function

Syntax: *button*

button returns a *boolean* value that indicates the state of the mouse button—*true* if the button is down, *false* if it is not.

Appendix C, "Predefined Graphics Procedures and Functions," contains examples that illustrate how to use the mouse in programs that incorporate graphics.

For example, you might write a program that performs a particular action until the user presses the mouse button. The outside loop would be

WHILE NOT *button* DO

The Getmouse Procedure

The *getmouse* returns the position of the mouse as x and y coordinates.

Syntax: *getmouse*(x, y)

For example, if the cursor appeared at the upperleft corner of the Drawing Window, *getmouse* would return 0,0.

The Paddlebutton Function

paddlebutton is a *boolean* function that returns *true* if the paddle button is down and *false* otherwise.

Syntax: *paddlebutton*(i);

Type: *boolean*

The *paddlebutton* function takes an integer parameter (0, 1, or 2) that is the number of the paddle to be read. You can also use this function to read the ⌘ and ◆ keys. In that case, the ⌘ key is button 0 and the ◆ key is button 1.

The Getpaddle Function

The *getpaddle* function takes a parameter of type *integer* that signifies the paddle number to be read. It must be 0, 1, 2, or 3. It returns a number between 0 and 255, which represents the rotation of the dial on the specified paddle.

Syntax: *getpaddle*(i);

The Sysbeep Procedure

Syntax: *sysbeep*(duration);

The *sysbeep* procedure sounds the Apple II system bell. The duration parameter is an integer value. Duration is measured in approximately .022 second increments.

For example, *sysbeep*(50); would sound the bell for approximately one second.

The Note Procedure

Syntax: *note*(frequency, reserved, duration)

The *note* procedure takes three integer parameters. The *note* procedure can be used to generate a variety of sounds by varying the frequency and duration of the note sounded. The first parameter determines frequency. See Table D-1 for the frequencies of approximately seven octaves of notes.

The third parameter determines the duration of the note. The parameter must be in the range 0..255. As with *sysbeep,* duration is measured in .022 second increments.

The second parameter must be supplied for compatibility with Macintosh Pascal. It must be a value in the range 0..255. If you run your program using Macintosh Pascal, the second parameter is used to determine the amplitude of the note.

Table D-1. Frequency and Duration

Note	Frequency by Octave							
B	62	123	247	494	988	1973	3946	
A#	58	117	233	466	932	1864	3743	
A	55	110	220	440	881	1761	3510	
G#	52	104	208	415	830	1663	3327	
G	49	98	196	392	784	1566	3142	
F#	46	92	185	370	740	1480	2959	
F	44	87	175	349	698	1398	2797	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1244	2495	4990
D	37	73	147	294	587	1176	2346	4713
C#	35	69	139	277	554	1109	2213	4426
C	33	65	131	262	523	1047	2095	4172
				Middle C				

Appendix E

The Standard Apple Numeric Environment

This appendix describes the Standard Apple Numeric Environment (SANE). It contains three parts:

- An introduction to SANE
- A discussion of the data types provided by SANE
- A description of each of the types, procedures, and functions contained in the SANE library.

SANE is the basis for all floating-point mathematical calculations performed by Instant Pascal. SANE meets all requirements for extended–precision floating-point arithmetic as prescribed by IEEE Standard 754. SANE ensures that all floating-point operations are performed consistently and that they return the most accurate results possible.

SANE provides an easy-to-use, flexible environment for floating-point calculations. It gives the programmer extremely accurate results without extra coding. You can write Pascal programs that use only the ANSI specified *real* type and be confident that your results are as accurate as possible within that format.

Programmers who are interested in precision beyond that possible using only the *real* type can use the other floating-point types provided as an extension to Pascal by SANE. In addition, the SANE library contains numerical functions not found in standard Pascal and routines for controlling the environment in which floating-point calculations are performed.

If you are using IP for advanced numerical programming, you might be interested in the complete and detailed description of SANE, which is contained in the *Apple Numerics Manual*, available from your Apple dealer.

What's in This Appendix

This appendix is divided into three parts:

- **Introduction**—The Introduction is a general overview of how computers handle numeric calculations and the kinds of problems you face when you use a computer for numerical programming. The Introduction describes how SANE provides solutions to these problems.

  You may be interested in this section, even if you don't think you'll ever need to use SANE's advanced features. A broad understanding of the IP real-types and how they can be used in numeric programming is good background for writing even elementary mathematical programs.
- **Description of the SANE Data Types**—This section contains a discussion of the real-types that are provided by SANE as an extension to ANSI Pascal.
- **Description of the SANE Library**—Not only does Instant Pascal use SANE as the basis for all floating-point arithmetic, IP also provides the SANE **library**, which includes types, procedures, and functions that are useful in numeric programming. The procedures and functions include financial, trigonometric, logarithmic, and exponential functions, and routines that allow you to customize the numeric environment to the special needs of your program.

Library: A set of constants, variables, types, procedures, and/or functions that are optionally available to a program.

An Introduction to SANE

The Standard Apple Numeric Environment is the set of methods that Instant Pascal uses for dealing with the problems inherent in performing many types of mathematical calculations with a computer.

Many people are used to the idea that numeric calculation is what computers do best. And that's true, with a few key reservations. Computers perform arithmetic very quickly; and when the calculations involve only moderate-sized integral values, they perform with complete accuracy.

However, when the calculations involve real values, computers are limited by the ways in which data is stored internally. These limitations can cause unexpected problems.

For example, if the computer arithmetic you're using isn't designed with a consistent set of rules, you may not be able to depend on the algebraic rules you might reasonably expect to work with your numeric operations. For

example, A + B may not equal B + A. Or you might find that A <> (is not equal to) B, but A − B = 0. These problems are caused by inconsistencies in how floating-point numbers are stored and manipulated in the computer.

SANE works in three ways to provide the environment needed for accurate and consistent floating-point calculations and for efficient data storage:

- By providing several data formats for floating-point values.
- By applying consistent rules for arithmetic operations and conversions between binary and decimal numeric data.
- By providing mechanisms that allow programs to monitor their numeric calculations for the occurrence of exceptional conditions.

To understand how SANE works, you first need a basic understanding of how computers represent numbers in memory.

The Internal Representation of Numbers

One of the major problems for the designers of computers and computer languages is simply deciding how to represent a floating-point number in the computer's memory.

People are used to representing numbers using ten symbols: the digits 0 through 9. This is the decimal, or base ten number system. The number of digits in a system is called the **radix** or **base** of the system. Most computers use the binary, or base two number system. The only symbols available are zero and one.

Decimal to Binary Conversion

Numeric constants appearing in programs, as well as those read from or written to the keyboard or a text file, are generally represented in the decimal number system. Program constants and numbers read as input must be converted to the computer's internal binary representation. And the opposite is also true—values stored in the computer in binary form must be converted to their decimal representation before being written to the screen or to a text file.

Positional Notation

Numbers with fractional parts are represented in binary notation in a familiar form: the integer and fractional parts are separated by a **radix point**. The term "radix point" is used rather than "decimal point" because the notation is independent of the number system.

Binary numbers are expressed in the same way that decimal numbers are, by using **positional notation**. The position of a digit, relative to the position of the radix point, determines the weight of the digit.

A decimal fraction, for example, 3.45, represents the value:

$3 * 1 + 4 * 1/10 + 5 * 1/100$

Using the binary system, the first number to the right of the radix point represents not tenths but halves. The second place represents quarters, the third eighths, and so on.

For example, the binary number 11.101_2 (the subscript indicates a binary number) equals

$1 * 2 + 1 * 1 + 1 * 1/2 + 0 * 1/4 + 1 * 1/8$

$= 2 + 1 + .5 + .125$

$= 3.625$

In this example, 3.625 can be converted exactly to a binary number. However, there are decimal values that do not have exact equivalents as terminating binary fractions: for example, one-tenth. When you convert one-tenth to a binary number you get this repeating fractional value:

$0.00011001100..._2$

There are an infinite number of binary digits to the right of the radix point. Because computers can't handle an infinite number of digits, this type of value must be converted to a representable value.

In general, computer conversions of decimal real numbers to binary (and vice versa) yield only approximate results. Since IP meets the stringent requirements of the IEEE Standard, the inaccuracies from binary-decimal conversion are minimal, except for unusually large and small numbers stored in the largest SANE format, the *extended* type.

Floating-Point Notation

A computer stores fractional numbers in binary form and represents them using **scientific** or **floating-point notation**. Using the example above, the number 3.65 can be represented with decimal floating-point notation as

Instant Pascal recognizes both the uppercase and lowercase letter "E" in floating-point notation.

.365e1

36.5e−1

365E−2

Using this notation, the number to the left of the letter "e" is the **significand**. The number to the right of the E is the **exponent**—the power of the base which multiplies the significand to get the correct result. This system allows the location of the radix point to change, which is why this is called floating-point notation.

Using binary notation, the exponent part is a power of two, rather than of ten. For example, the number 11.101_2 can be written

$.11101_2 * 2^2$
$1.1101_2 * 2$
$11101_2 * 2^{-3}$

SANE (and IP) represents real-type numbers in computer memory using a variation of this system. The radix point and the base aren't necessary—each format is designed with one bit for the sign and with a predetermined number of bits for the exponent part and the significand.

Data Formats

A data format is a template that determines how many bits of memory will be used for each piece of data of a particular type.

When you write a computer program you create variables to hold the values that the program will be operating on. One of the tasks of a high-level programming language, such as IP, is to allocate an appropriate amount of computer memory for each variable in a program. The amount of memory depends upon what type of data is being stored.

It's relatively easy to store data in the form of characters. Every character in the ASCII character set can be represented using eight bits. Any variable of type *char* will always be given eight bits of memory.

Representing floating-point numbers in memory is more complicated. As mentioned above, SANE formats used for floating-point values are divided into three parts: a part to represent the sign (plus or minus), a part to represent the exponent, and a part to represent the significand. The sign can be represented with only one bit—it's on, or it's off.

The number of bits given to the significand determines the **precision** of the number stored. The SANE *extended* type uses 64 bits to represent the significand. This allows the storage of 19 or 20 decimal digits.

The number of bits given to the exponent determines the **range** of the values you will be able to represent in that format. For example, the SANE type *extended* allocates 15 bits for the exponent. The exponent range is −16383..16383, giving a range of values for the *extended* type of about 1.9e−4951 to 1.1e+4932.

Extended Arithmetic

While the IP types *real*, *double*, and *comp* are intended for economical data storage, the *extended* type is the foundation for all arithmetic computation. As specified by the IEEE Standard, all basic arithmetic operations, including addition, subtract, multiply, divide, and square root, yield the best possible results. In IP these operations produce *extended* results, so they are accurate to a precision of 19 decimal digits, throughout a range exceeding 10^{-4900} to 10^{+4900}

IP takes advantage of extended arithmetic by storing all non-integer numeric constants in the *extended* format, and by evaluating all non-integer numeric expressions to *extended*, regardless of the types involved. For example, the entire right side of the assignment below will be computed in *extended* before being converted to the type of the left side:

```
VAR
 x, a, b, c: real;

BEGIN
 .
 .
 .
 x:=(b+sqrt(b*b-a*c))/a;
```

With no special effort by the programmer, IP performs computations using extended precision and range. Extra precision means smaller roundoff errors, so that results are more accurate, more often. Extra range means overflow and underflow are rarer, so that programs work more often.

By following a few simple programming practices you can exploit the *extended* type, beyond what IP does for you automatically.

Declare variables used for intermediate results to be of type *extended*. This practice is illustrated in the following example, which computes a sum of products.

```
VAR
 Sum: real;
 X, Y: ARRAY[1..N] OF real;
 I: integer;
 T: extended;  {To hold intermediate results}
BEGIN
 .
 .
 .
 T:=0.0;
 FOR I:= 1 TO N DO
  T:=T+X[I]*Y[I];
 Sum:=T;
```

Had T been declared *real*, like the input arrays X and Y and the result Sum, each time through the loop the assignment to T would have caused a roundoff error at the limit of single precision. In the example, all roundoff errors are at the limit of extended precision, except for the one caused by the assignment of T to Sum. This means roundoff errors will be less likely to accumulate to produce an inaccurate result.

Declare formal value parameters and function results to be of type *extended*, rather than *real*, *double*, or *comp*. This saves IP from having to do unnecessary conversions between numeric types, which may result in loss of accuracy. The example below illustrates this practice.

```
FUNCTION Area(Radius:extended):extended;
BEGIN
 Area:=Pi*Radius*Radius
END;
```

Special Cases

Although use of the *extended* type makes programs work more often, exceptional cases do arise. Your programs may contain statements like these:

```
Average:=Sum/Count;
Area:=Side*Side;
```

where all the variables are of type *real*. What happens if Count is zero, if Count and Sum are both zero, or if the product of Side*Side is too large to be represented in the *real* format? Normally, IP will stop your program, as prescribed by ANSI Pascal, and tell you that you have made an error.

Default: A value, action, or setting that is assumed or set in the absence of explicit instructions otherwise.

Halting your program when this type of "error" occurs is not the only way IP can work. Instead, IP can assign special values to Average and Area, and your program can continue. In fact, the IEEE Standard refers to "exceptions" rather than "errors" and specifies "no halts" as the **default** mode of operation for its arithmetic. To install the IEEE defaults you would use this statement:

```
SetEnvironment(0);
```

The SetEnvironment procedure is described in detail later in this appendix.

The SANE library also contains functions and procedures for determining when exceptional cases occur.

Number Classes

Representations in the SANE data formats fall into five classes:

- Normalized numbers—like 3.0, 75.8, −2.3e78 and all others that can be represented with a leading significand bit of 1.
- Zero—+0 and −0.
- Infinities—positive and negative infinity.
- NaNs—short for Not-a-Number.
- Denormalized numbers—nonzero numbers that are too small for normalized representation.

Infinities

Infinities are special SANE representations that can arise in two ways from operations on finite values:

- When a operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity.
- When an operation produces a number with magnitude too great for the number's intended floating-point format, the result may (depending on the current rounding direction) be an infinity.

IP predefines a constant (INF) to have the value positive infinity. Also, INF represents infinity for input and output of floating-point values. Infinities behave like mathematical infinities. For example, 1 − INF = − INF. Infinities can be helpful even when "infinity arithmetic" is not the goal. For example, if X * X is too large for the *extended* format, the expression 1 + 1/(X * X) still computes to the correct value of 1 (assuming overflow halts are off).

Try this:

```
PROGRAM UseInf;
USES SANE;
VAR
 X:extended;
BEGIN
 SetEnvironment(0);
 X:=1e4000;
 writeln('X*X=',X*X);
 writeln('1/(X*X)=',1/(X*X));
 writeln('1+1/(X*X)=',1+1/(X*X));
END.
```

NaNs

Another special SANE representation is a NaN (Not-a-Number). A NaN is produced whenever an operation cannot produce a meaningful numeric result. For example, 0/0 and sqrt(−1) yield NaNs.

Each time a NaN is generated, an associated NaN code is returned as part of the NaN's representation. This code tells you what kind of operation caused the NaN to be created. NaN codes, shown in Table E-1, can help with debugging.

Table E-1. NaN Codes

Code	Meaning
1	Invalid square root, such as sqrt(−1)
2	Invalid addition, such as (+INF) − (+INF)
4	Invalid division, such as 0/0
8	Invalid multiplication, such as 0 * INF
9	Invalid remainder or MOD, such as X MOD 0
17	Attempt to convert invalid ASCII string
20	Result of converting the *comp* NaN to floating-point format
21	Attempt to create a NaN with a zero code
33	Invalid argument to trig routine
34	Invalid argument to inverse trig routine
36	Invalid argument to log routine
37	Invalid argument to x^t or x^y routine
38	Invalid argument to financial function
255	Uninitialized storage (signalling NaN)

The statement writeln(0/0) will produce the result NAN(004) (provided the invalid operation halt is off). NAN(004), nan(4), and NaN are examples of acceptable input for reading a NaN into a SANE variable.

Denormalized Numbers

Whenever possible, SANE stores values in normalized form: the most significant bit of the significand is a one, rather than a zero.

However, when a very small number is being stored, and the exponent is the smallest possible negative value, it is possible to store still smaller values by storing leading zeroes. For example,

$1.0..0_2 * 2^{-126}$ — smallest normalized *real*

$0.1..0_2 * 2^{-126}$ — still smaller denormalized *real*

Because of denormalized numbers, IEEE arithmetic has the desirable property that A<>B if and only if A − B <> 0. In most non-IEEE arithmetics, A − B will "flush to zero" if A−B is too small for normalized representation, even though A and B may be different values.

Exceptional Conditions

Exceptional conditions can arise from floating-point calculations in a number of cases. For example, multiplying two very large values can result in a value too large to be represented in one of the IP data formats. Or an operation such as 0/0 can be performed.

SANE provides a way for a program to determine when a floating-point calculation has resulted in one of these exceptional conditions. Exceptional conditions fall into five categories:

- invalid operation
- underflow
- overflow
- divide-by-zero
- inexact

Invalid Operation

The invalid operation exception arises when operands for an operation are invalid, so that a meaningful numeric result is impossible. For example, 0/0 and sqrt(−1) are invalid operations.

Underflow

Underflow occurs when a result is both denormalized and has lost significant digits through rounding. For example, to return the result of:

$(1.00000000000000000000001_2 * 2^{-126}) / 2$

to the real format, a leading zero would be introduced and the last significant bit would be lost in rounding. This result:

$0.10000000000000000000000_2 * 2^{-126}$

would be returned and underflow would be signalled.

Overflow

The condition of calculating a value that is too large to fit in the format of its designated type is called **overflow**. The destination format must be one of the floating-point types; if the destination format is an integral type, the invalid exception occurs.

Divide-by-Zero

The divide-by-zero exception occurs when a finite nonzero number is divided by zero. It also occurs when an operation on finite operands produces an exact infinite result. For example, the operation 1/0 (which results in INF) and the operation ln(0) (which results in −INF) both signal divide-by-zero.

Inexact

The inexact exception occurs if the rounded result of an operation is not identical to the mathematical (exact) result. (Thus, any time overflow or underflow occurs, the inexact exception is signaled.) For example, the operation 2/3 signals inexact, regardless of the floating-point format used.

The Environment

The SANE "environment" consists of :

- rounding direction
- rounding precision
- exception flags
- halt settings

The SANE library includes procedures and functions that allow you to determine the current status of the environment. These procedures and functions can be used to flag exceptional conditions and to control optional environment settings. For example, you may be working with very small values and need to know exactly when underflow occurs. Or you might want to have floating-point conversions rounded downward.

The SANE Data Types

The original specification for Pascal called for only one data type for use with floating-point numbers—the *real* type. Instant Pascal supplements the *real* type with three others: *double*, *extended*, and *comp*.

A Note on Terminology

SANE is designed to be a generic system that can be used with a variety of high-level languages. SANE provides the three floating-point types specified by the IEEE Standard (where they are called *single*, *double*, and *extended*). The ANSI Standard specifies that any implementation of Pascal that meets its requirements must include a type named *real*. IP uses the SANE type *single* as the ANSI type *real*. The body of this manual refers to the *real* type, while the same type is referred to as *single* in this appendix. The IP interpreter will accept either of these terms and treats them both in exactly the same way.

Descriptions of the Types

The *single* type is the smallest format for use with floating-point numbers. It stores floating-point numbers using 32 bits of storage.

The *double* type is twice the size of the *single* type. It uses 64 bits for storage.

The *extended* type is larger yet—it uses an 80-bit format. All arithmetic involving real-type values is done using the *extended* type.

The *comp* type stores integral values in a 64-bit format. It's classified as a real-type because arithmetic done with operands of type *comp* uses the *extended* type. Results assigned to a variable of type *comp* are converted from *extended*.

Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- fixed- or floating-point form
- precision
- range
- memory usage

The precision, range, and memory usage for each SANE data type are shown in Table E-2.

Many programs require a counting type that counts things (pennies, dollars, widgets) exactly. Using SANE, you can write a program that deals with monetary values by representing these values as integral numbers of cents or mils, which can be stored exactly in the *comp* type. The sum, difference, or product of any two *comp* values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, *comp* values (for example, the results of accounting computations) can be mixed with *extended* values in floating-point computations (such as compound interest).

Arithmetic with *comp* type variables, like all SANE arithmetic, is done internally using the *extended* type for arithmetic. There is no loss of precision, as conversion from *comp* to *extended* is always exact. You can save by storing numbers in the *comp* type, which is 20 percent shorter than extended (64 versus 80 bits).

Values Represented

The floating-point types (*single*, *double*, and *extended*) store binary representations of a **sign** (+ or −), an **exponent**, and a **significand**. A represented number has the value

$$\pm \text{significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

Range and Precision of SANE Types

The range and precision of the real-types supported by SANE and IP are shown in Table E-2. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table E-2. SANE Types

Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	——	-16383
Maximum	127	1023	——	16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7—8	15—16	18—19	19—20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	≅-9.2E18	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	≅9.2E18	1.1E+4932
Infinities	Yes	Yes	No	Yes
NaNs	Yes	Yes	Yes	Yes

Example

Using the *single* type, the largest representable number has

$$\text{significand} = 2 - 2^{-23}$$

$$= 1.11111111111111111111111_2$$

$$\text{exponent} = 127$$

value $= (2 - 2^{-23}) * 2^{127}$

$\cong 3.403 * 10^{38}$

the smallest representable positive normalized number has

significand $= 1$

$= 1.00000000000000000000000_2$

exponent $= -126$

value $= 1 * 2^{-126}$

$\cong 1.175 * 10^{-38}$

and the smallest representable positive denormalized number has

significand $= 2^{-23}$

$= 0.00000000000000000000001_2$

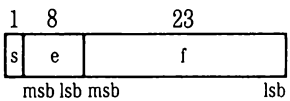
exponent $= -126$

value $= 2^{-23} * 2^{-126}$

$\cong 1.401 * 10^{-45}$

The Single Type

A 32-bit *single* number is divided into three fields as shown below.



The value v of the number is determined by these fields:

If $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$.

If $e = 0$ and $f \neq 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$.

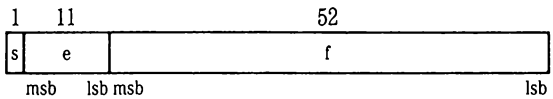
If $e = 0$ and $f = 0$, then $v = (-1)^s * 0$.

If $e = 255$ and $f = 0$, then $v = (-1)^s * \infty$.

If $e = 255$ and $f \neq 0$, then v is a NaN.

The Double Type

A 64-bit *double* number is divided into three fields as shown below.

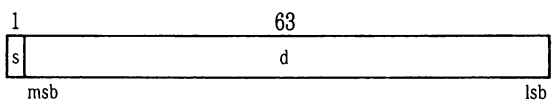


The value v of the number is determined by these fields:

If $0 < e < 2047$,	then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$,	then v is a NaN.

The Comp Type

A 64-bit *comp* number is divided into two fields as shown below.

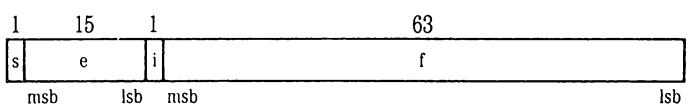


The value v of the number is determined by these fields:

If $s = 1$ and $d = 0$,	then v is the unique comp NaN.
Otherwise,	v is the two's-complement value of the 64-bit representation.

The Extended Type

An 80-bit *extended* format number is divided into four fields as shown below.



The value v of the number is determined by these fields:

If $0 <= e < 32767$, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$.

If $e = 32767$ and $f = 0$, then $v = (-1)^s * \infty$, regardless of i.

If $e = 32767$ and $f \neq 0$, then v is a NaN, regardless of i.

The SANE Library

This section explains each of the constants, types, functions, and procedures contained in the SANE library.

Descriptions of Constants and Types

Each of the constants and types defined by the SANE library is briefly discussed in the following section. For more information, see the descriptions of the specific procedures and functions mentioned.

The DecStrLen Constant

DecStrLen is defined by this declaration:

```
DecStrLn = 255;
```

DecStrLen (Decimal String Length) is the constant that defines the maximum length of a decimal numeric string. DecStrLen is the size attribute of variables of type DecStr.

Exception Condition Constants

These declarations define the exception condition constants:

```
Invalid = 1;
Underflow = 2;
Overflow = 4;
DivByZero = 8;
Inexact = 16;
```

These constants are used to define the value of a variable of the Exception type.

For example, if e is a variable of type *Exception*, then

e := Invalid + Overflow + DivByZero

gives e a value that represents these three exceptions collectively.

The *SetException* and *SetHalt* procedures each take arguments of the type *Exception*.

The *TestException* and *TestHalt* functions each return a value of the type *Exception*.

The DecStr Type

This declaration defines the DecStr type:

```
DecStr = STRING;
```

The DecStr type is a string with a size attribute of DecStrLen, or 255 characters. It's used to hold the decimal representation, in ASCII characters, of a number.

The DecForm Record Type

A record of type *DecForm* (Decimal Format) is defined by this declaration:

```
DecForm = RECORD
 Style :  (FloatDecimal, FixedDecimal);
 Digits :  integer;
END;
```

A *DecForm* record holds the specifications for the format of a decimal number.

- The *Style* field determines whether the decimal representation will be floating-point or fixed-point.
- The *Digits* field holds the number of significant digits for float style or the number of digits to the right of the decimal point for fixed style.

The *Num2Str* procedure takes a *DecForm* argument. It uses the information in *DecForm* to determine the format for the string that's returned by the procedure.

The RelOp Type

The *RelOp* (Relational Operator) type is defined by this declaration:

```
RelOp = (GreaterThan, LessThan, EqualTo, Unordered);
```

A result of this type is returned by the *Relation* function, described later.

The NumClass Type

The NumClass type is defined by this declaration:

```
NumClass = (SNaN, QNaN, Infinite, ZeroNum,
            NormalNum, DenormalNum);
```

Number Class	Meaning
SNaN	Signaling NaN
QNaN	Quiet NaN
Infinite	Infinity or −Infinity
ZeroNum	0 or −0
NormalNum	Normalized number
DenormalNum	Denormalized number

Quiet NaNs are the usual kind produced by floating-point operations. Signaling NaNs, potentially useful for flagging uninitialized variables, are discussed in the *Apple Numerics Manual.*

The *NumClass* type is used to return results from the inquiry functions, described below.

The Exception Type

A variable of type *Exception* holds an integer value that corresponds to the value of one of the *Exception* constants, or to a sum of two or more of the *Exception* constants. The *Exception* type is defined by this declaration:

```
Exception = integer;
```

The *SetException* and *SetHalt* procedures each take arguments of the type *Exception.* The *TestException* and *TestHalt* functions each return a value of the type *Exception.*

The RoundDir Type

The *RoundDir* (Rounding Direction) type is defined by this declaration:

```
RoundDir = (ToNearest, Upward, Downward, TowardZero);
```

The *RoundDir* type is used to determine how values are to be rounded, when rounding becomes necessary during arithmetic operations or conversions. The *SetRound* procedure takes an argument of type *RoundDir*. The *GetRound* function returns a value of type *RoundDir*.

The RoundPre Type

The *RoundPre* (Rounding Precision) type is defined by this declaration:

```
RoundPre = (ExtPrecision, DblPrecision,
            RealPrecision);
```

Rounding precision can be used to simulate arithmetic with only single or double precision. The *SetPrecision* procedure takes an argument of type *RoundPre*. The *GetPrecision* function returns a value of type *RoundPre*.

The Environment Type

A variable of type *Environment* holds a value that represents the settings of the SANE environment. For example, a setting of 0 represents the default IEEE setting (including no halts set). The *Environment* type is defined with this declaration:

```
Environment = integer;
```

You use a variable of type *Environment* with these environmental access routines: *SetEnvironment*, *GetEnvironment*, *ProcEntry*, and *ProcExit*.

Predefined Numeric Procedures and Functions

This section includes a description of each of the procedures and functions in the SANE library. More detailed information can be found in in the *Apple Numerics Manual*, which is available from Apple dealers.

Remember that any function with a formal parameter of any of the real-types can be passed a value of any real- or integer-type.

Conversions Between Numeric Binary Types

The SANE library contains functions that convert numeric values (in binary representation) to the binary formats of the *integer*, *longint*, and *extended* types.

The Num2Integer and Num2Longint Functions

```
FUNCTION Num2Integer(x : extended):integer;

FUNCTION Num2Longint(x : extended):longint;
```

The *Num2Integer* function takes an extended argument and returns a result of type *integer*.

The Num2Longint function takes an extended argument and returns a result of type *longint*.

The value returned by these functions depends upon the rounding direction (set using the *RoundDir* procedure). Using the standard rounding direction, ToNearest, these examples

```
Num2Integer(99.6);
Num2Longint(99.6);
```

return the value 100.0.

Num2Integer and Num2Longint are similar to the IP functions *round* and *trunc*. However, *Num2Integer* and *Num2Longint* take the current rounding direction into account. The *round* function always returns the nearest *longint* value; the *trunc* function always rounds toward zero.

Here's an example showing the differences in these functions:

```
VAR
 A : extended;
 B, C, D : longint;

BEGIN
 A := 99.999;
 B := Num2Longint(A);
 C := round(A);
 D := trunc(A);
END;
```

After this code is executed, both B and C would have the value 100 and D would have the value 99.

```
BEGIN
 A := 99.999;
 SetRound(Downward);
 B := Num2Longint(A);
 C := round(A);
 D := trunc(A);
END;
```

However, using this code, the value of B and D would be 99. But the value of C would again be 100. The *round* and *trunc* functions always calculate their value in the same way, regardless of the rounding direction.

Using the ToNearest rounding direction, Num2Integer and Num2Longint round values halfway between two integers to the nearest even integer (as prescribed by the IEEE Standard). For example, Num2Integer(2.5) returns 2. The *round* function rounds these halfway values away from zero. For example, *round*(2.5) returns 3.

The Num2Extended Function

```
FUNCTION Num2Extended(x : extended):extended;
```

The *Num2Extended* function can be passed any real-type or integer-type argument. It converts its argument to the *extended* format. This is useful for forcing floating-point arithmetic when all variables involved are of the integer types.

Conversions Between Decimal String and Binary

The SANE library includes the *Num2Str* procedure and the *Str2Num* function to convert numbers between decimal ASCII character representations and binary.

Important The IP input and output procedures, described in Chapter 10, perform all necessary conversions between decimal ASCII and binary.

The Num2Str Procedure

```
PROCEDURE Num2Str(f:DecForm; x:extended; VAR s:
DecStr);
```

The procedure converts an *extended* value x to a decimal string, returned in s, using the specifications in the DecForm record f. Here are some examples of how *Num2Str* uses the arguments passed to it to format a string.

DecForm.Style	DecForm.Digits	x	s
FloatDecimal	6	123.45	'1.23450e+2'
FloatDecimal	2	123.45	'1.2e+2'
FixedDecimal	6	123.45	'123.450000'
FixedDecimal	2	123.45	'123.45'

The Str2Num Function

```
FUNCTION Str2Num(s:DecStr):extended;
```

The *Str2Num* function takes a decimal string argument (of type DecStr) and converts it to type *extended*.

Arithmetic

The SANE library includes a set of functions that supplement the arithmetic functions described in Chapter 3.

The Remainder Function

```
FUNCTION Remainder(x,y:extended; VAR quo:integer);
```

The *Remainder* function returns the remainder of the division of its two extended arguments x/y, as specified by the IEEE Standard. This function returns an exact remainder of the smallest possible magnitude. The result is computed as:

x−n*y

where n is a nearest integral approximation to the quotient x/y. For example, Remainder(9,5,q) returns −1, because −1 = 9−2*5.

The integer variable argument quo receives the seven low-order bits of n as a value between −127 and 127; this is useful for programming functions, like the trigonometric functions, that require argument reduction.

Important Remember that the Pascal operator MOD can be used only with integral values. The *Remainder* function is used with real-type or integer-type values.

The Rint Function

```
FUNCTION Rint(x:extended):extended;
```

The *Rint* function takes an *extended* argument and rounds it to an integral value in the *extended* format. Note that all sufficiently large floating-point values are integral. The result depends upon the rounding direction, which can be changed using the *SetRound* procedure.

The Scalb Function

```
FUNCTION Scalb(n:integer; x:extended):extended
```

The *Scalb* function takes two arguments. The first is a value of type *integer*, the second is an *extended* value. The function scales the *extended* value by the power of two specified by the *integer* argument. The value $2^n X$ is returned in *extended* format.

The Logb Function

```
FUNCTION Logb(x:extended):extended;
```

The *Logb* function takes an *extended* argument and returns the largest power of two that does not exceed its argument's magnitude. For example,

```
Logb(-65535)
```

yields 15 because $2^{15} <= 65535 < 2^{16}$.

The CopySign Function

```
FUNCTION CopySign(x,y:extended):extended;
```

The *CopySign* function takes two *extended* arguments. It returns the second argument, but with the sign of the first argument. For example, CopySign(2.0, −3.0) yields 3.0. The CopySign function returns an extended value.

The NextReal Function

```
FUNCTION NextReal(x,y:real):extended;
```

The *NextReal* function takes two *real* arguments. It returns the next value that can be represented in the *real* format after the first argument, in the direction of the second argument. It returns an *extended* value.

The NextDouble Function

```
FUNCTION NextDouble(x,y:double):extended;
```

The *NextDouble* function takes two *double* arguments. It returns the next value that can be represented in the *double* format after the first argument, in the direction of the second argument. It returns an *extended* value.

The NextExtended Function

```
FUNCTION NextExtended(x,y:extended):extended;
```

The *NextExtended* function takes two *extended* arguments. It returns the next value that can be represented in the *extended* format after the first argument, in the direction of the second argument. It returns an *extended* value.

The Log2 Function

```
FUNCTION Log2(x:extended):extended;
```

The *Log2* function takes an *extended* argument and returns the base-2 logarithm of its argument in *extended* format.

The Ln1 Function

```
FUNCTION Ln1(x:extended):extended;
```

The *Ln1* function takes an argument of type *extended* and returns the base-e logarithm of 1 plus the argument: ln(1+X). It returns an *extended* value. For X near 0, Ln1(X) is more accurate than Ln(1.0+X).

The Exp2 Function

```
FUNCTION Exp2(x:extended):extended;
```

The *Exp2* function takes and *extended* argument and returns 2 raised to the power of the argument: 2^x. It returns an *extended* value.

The Exp1 Function

```
FUNCTION Exp1(x: extended):extended;
```

The *Exp1* function takes an *extended* argument x and returns $e^x - 1$. It returns an extended value. For x near 0, Exp1(x) is more accurate than Exp(x) − 1.0.

The XpwrI Function

```
FUNCTION XpwrI(x:extended; i:integer):extended;
```

The *XpwrI* function takes one *extended* argument x and one *integer* argument i. It returns the value of the extended argument, raised to the power specified by the *integer* argument: x^i. It returns an extended value.

The XpwrY Function

```
FUNCTION XpwrY(x,y:extended):extended;
```

The *XpwrY* function takes two *extended* arguments, x and y. It returns the value of the first argument, raised to the power specified by the second argument: x^y. It returns an *extended* value.

Financial Functions

SANE provides two functions that can be used in financial applications: the *Compound* function and the *Annuity* function.

The Compound Function

```
FUNCTION Compound(r,n: extended):extended;
```

The *Compound* function takes two *extended* arguments. The first argument specifies the interest rate; the second specifies the number of periods. It returns $(1+r)^n$, which is the principal plus accrued compound interest on an original investment of one unit. It returns an *extended* value.

The Annuity Function

```
FUNCTION Annuity(r,n: extended):extended
```

The *Annuity* function takes two *extended* arguments. The first argument specifies the interest rate; the second specifies the number of periods. It returns $(1-(1+r)^{-n})/r$, which is the present value factor of an ordinary annuity. It returns an *extended* value. Here is an example of how the *Annuity* function can be used:

```
PROGRAM Loan;
VAR
 Loan, Payment, Interest, Periods:  extended;
BEGIN
 writeln('Loan amount: ');
 readln(Loan);
 writeln('Annual interest rate (Enter as a decimal.): ');
 readln(Interest);
 writeln('Number of years:  ');
 readln(Periods);
 Payment:= Loan/Annuity(Interest/12, Periods*12);
 write('Your payment is:  ');
 write(Payment:8:2);
END.
```

In this example, given a loan amount of $120,000 and an interest rate of 10.75% for 30 years, the payment will be 1120.18.

Trigonometric Functions

Instant Pascal includes predefined *sin*, *cos*, and *arctan* functions. In addition, the SANE library provides the *Tan* function.

The Tan Function

```
FUNCTION Tan(x:extended):extended;
```

The *tan* function returns the tangent of an *extended* argument. In a right triangle, Tan(x) is the ratio of the length of the side opposite an angle of x radians to the length of the side adjacent to it. Note that the argument must be expressed in radians. The Tan function returns an *extended* value.

Inquiry Functions

The library includes several functions that allow you to determine the class of a numeric value. The result of each of these functions is of type *NumClass*, described above. In addition, they include a function that returns the sign of a numeric value.

The ClassReal Function

```
FUNCTION ClassReal(x:real):NumClass
```

The *ClassReal* function determines the number class of its *real* argument. For example,

```
ClassReal(1)
ClassReal(1e-310)
```

The first function call returns NormalNum, the code for a normalized number. The second call returns ZeroNum, the code for zero (because 1e−310 rounds to +0 in the *real* format).

The ClassDouble Function

```
FUNCTION ClassDouble(x:double):NumClass;
```

The *ClassDouble* function determines the number class of its *double* argument. The result is of type *NumClass*. For example,

```
ClassDouble(0.0/0.0)
ClassDouble(1e-310)
```

The first example returns QNaN, the code for a quiet NaN. The second example returns DenormalNum, the code for a denormalized number (because 1e−310 is denormalized in the *double* format).

The ClassExtended Function

```
FUNCTION ClassExtended(x:extended):NumClass;
```

The *ClassExtended* function determines the number class of its *extended* argument. The result is of type *NumClass*. For example,

```
ClassExtended(1/0)
ClassExtended(e-310)
```

The first example returns Infinite, the code for infinities. The second example returns NormalNum, the code for a normalized number.

The ClassComp Function

```
FUNCTION ClassComp(x:comp):NumClass;
```

The *ClassComp* function determines the number class of its *comp* argument. The result is of type *NumClass*. For example,

```
ClassComp(1)
ClassComp(0.1)
```

The first example returns NormalNum, the code for a normal number. The second example returns ZeroNum, the code for zero. (Remember that *comp* stores integral values.)

The SignNum Function

```
FUNCTION SignNum(x:extended):integer;
```

The *SignNum* function takes an argument of type *extended* and returns an integer value that indicates the sign of the argument. The value returned is one if the argument's sign is negative, 0 if the argument's sign is positive.

The RandomX Function

```
FUNCTION RandomX(VAR x:extended):extended;
```

Instant Pascal also includes a *random* function, which returns a random number of type *longint*. This function is described in Chapter 3.

The *RandomX* function takes a variable argument of type *extended* which contains an integral value in the range $1 <= r <= 2^{31} - 2$. It returns the next random number (in *extended* format) in sequence within the same range. The variable argument is updated to the value returned. RandomX uses this algorithm:

$NewX = (7^5 * OldX) \text{ MOD } (2^{31} - 1)$

The NaN Function

```
FUNCTION NAN(x:integer):extended;
```

The *NAN* function takes an integer argument and returns a NaN, in *extended* format, associated with the code given as an argument. The SANE NaN error codes are shown in Table E-1 in the preceding section, "NaNs."

The Relation Function

```
FUNCTION Relation(x,y: extended):RelOp;
```

The *Relation* function takes two *extended* arguments and returns a value of type *RelOp*. The value returned specifies the relationship between the two arguments.

For example,

```
Relation(0.1, NAN(0));
```

returns Unordered, as all comparisons involving NaNs are unordered.

Environmental Access Procedures and Functions

The SANE environment access routines allow you to determine how calculations are to be performed and how to respond to exceptional conditions. The environment consists of:

- the rounding direction
- the rounding precision
- exception flags
- halt settings

The Rounding Direction

The rounding direction can be set in four ways:

- to nearest
- upward
- downward
- toward zero

The standard rounding direction is to nearest. You can find out what the current rounding direction by using the *GetRound* function. You can change the rounding direction by using the *SetRound* function.

The GetRound Function

```
FUNCTION GetRound:RoundDir;
```

The *GetRound* function returns the current rounding direction as a value of type *RoundDir*.

The RoundDir type is discussed in the previous section on predefined SANE types and constants.

The SetRound Procedure

```
PROCEDURE SetRound(r:RoundDir);
```

The *SetRound* procedure takes an argument of type *RoundDir*. The procedure sets the effective rounding direction to the one indicated by the argument.

For example, the code below saves the current rounding direction, computes a function using TowardZero rounding, and finally restores the saved rounding direction.

```
VAR
 R: RoundDir;
 X, Y: extended;

BEGIN
 .
 .
 .
 R:=GetRound;
 SetRound(TowardZero);
 Y:=f(x);
 SetRound(R);
```

The Rounding Precision

You may find that you want to use SANE to perform calculations and then to simulate the results you would get if you used a system that did not provide extended precision arithmetic. Normally, all IP floating-point calculations return results that are rounded to extended precision and range. However, the rounding precision can be set to *single* or *double* precision and range. Results will still be returned in the *extended* format.

There is no performance benefit in setting *single* or *double* rounding precision. You can access the rounding precision by using the SetPrecision procedure and the GetPrecision function.

The GetPrecision Function

```
FUNCTION GetPrecision:RoundPre;
```

The *GetPrecision* function returns a value of type *RoundPre* which indicates the current rounding precision.

The SetPrecision Procedure

```
PROCEDURE SetPrecision(p:RoundPre);
```

The *SetPrecision* procedure takes an argument that is a variable of type RoundPre, which indicates the desired rounding precision.

Exceptions

A numeric calculation can result in one of five kinds of exceptional conditions. Whenever one of these exceptional conditions occurs, a flag is set. The SANE library defines a constant for each kind of exception.

Exception	Constant Value	Event Causing Exception	Example
Invalid	1	Operation not meaningful—NaN result	sqrt(-1)
Underflow	2	Accuracy lost—result too small	$2^{-16383}/3$
Overflow	4	Result too large for number representation	Exp2(16384)
DivByZero	8	Division of non-zero number by zero	1/0
Inexact	16	Rounded result not same as exact math result	1/3

- If an invalid operation is performed, the invalid-operation flag is set.
- If underflow occurs, the underflow flag is set.
- If overflow occurs, the overflow flag is set.
- If a division by zero occurs, the divide-by-zero flag is set.
- If the result of the calculation is inexact, the inexact flag is set.

The SetException Procedure

```
PROCEDURE SetException(e:Exception; b:boolean);
```

The *SetException* procedure takes one argument of type *Exception* and a second argument of type *boolean*. If the second argument is *true*, the procedure signals the exceptions encoded in its first argument. If the second argument is *false*, it clears the exceptions flags specified by the first argument. For example,

```
SetException(Overflow + Inexact, true);
```

This statement signals the Overflow and Inexact exceptions. If halt on Overflow or Inexact were set, this statement would halt the program.

The TestException Function

```
FUNCTION TestException(e:Exception):boolean;
```

The *TestException* function takes an argument of type Exception and returns a *boolean* value that indicates whether any of the exceptions encoded in its argument are set or not.

Following the statement in the "SetException" section above, the statement:

```
TestException(OverFlow + Invalid);
```

would return *true*.

Using Exceptional Conditions to Halt a Program

The SANE environment includes a halt setting for each of the five exceptions that determines whether the occurrence of the exception halts the program. By default, IP sets halt on Invalid, Overflow, and DivByZero. The IEEE Standard default calls for all halts clear (off).

You can access the halt settings by using the *TestHalt* Function and the *SetHalt* procedure.

The TestHalt Function

```
FUNCTION TestHalt(e:Exception):boolean;
```

The TestHalt function takes an argument of type *Exception* and returns a *boolean* value. If a halt is enabled for any of the exceptions indicated by the argument, the function returns *true*. If none of those halts are enabled, it returns *false*.

The SetHalt Procedure

```
PROCEDURE SetHalt(e:Exception; b:boolean);
```

The *SetHalt* procedure takes two arguments. The first is of type *Exception.* This indicates which exceptions you want to halt your program. The second argument is of type *boolean.* If the value of the *boolean* argument is *true*, occurrences of the indicated exceptions will cause your program to halt. If it is false your program will continue to run when these exceptions occur.

Saving and Restoring Environmental Settings

The entire SANE environment (rounding direction, rounding precision, exception flags, and halt settings) can be encoded in a value of type *Environment.* The procedures described below access the current SANE environment as a whole. They are useful for managing the environment so that routines run with the environments they require and for controling the exception information passed between routines.

The GetEnvironment Procedure

```
PROCEDURE GetEnvironment(VAR e:Environment);
```

The *GetEnvironment* procedure takes an argument of type *Environment* as a variable parameter and assigns the current settings of the environment to that variable.

When your program begins the environment will reflect the IP defaults:

- Rounding direction —ToNearest
- Rounding Precision —Extended
- All exception flags cleared
- Halt on Invalid, Underflow, and DivByZero

The SetEnvironment Procedure

```
PROCEDURE SetEnvironment(e:Environment);
```

The *SetEnvironment* procedure takes an argument of type *Environment.* It sets the effective environment to the one encoded in its argument. To install the IEEE standard defaults, use the statement:

```
SetEnvironment(0);
```

The following procedure ensures that it will run under the IEEE default environment, while not affecting its caller's environment:

```
PROCEDURE P;
VAR
 SaveEnv:Environment;
BEGIN
 GetEnvironment(SaveEnv);
 SetEnvironment(0);
 .
 .
 .
 SetEnvironment(SaveEnv);
END;
```

The ProcEntry Procedure

```
PROCEDURE ProcEntry(VAR e:Environment);
```

The *ProcEntry* procedure saves the current environment (the rounding direction, rounding precision, exception flags, and halt settings) in the Environment variable passed to the procedure, and then sets the environment to the IEEE defaults.

The statement

```
ProcEntry(e);
```

is equivalent to

```
GetEnvironment(e);
SetEnvironment(0);
```

The ProcExit Procedure

```
PROCEDURE ProcExit(e);
```

The *ProcExit* procedure takes an argument of type *Environment*. It temporarily saves the current exception flags, sets the effective environment to be the one encoded in its argument, and then signals the temporarily saved exceptions.

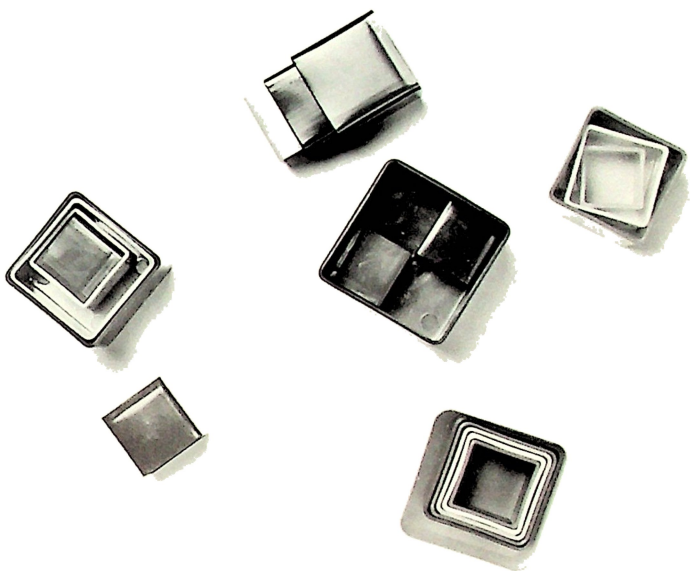
ProcEntry and ProcExit can be used in routines to selectively hide spurious exceptions from the routine's caller, as shown in the following example.

```
FUNCTION ArcCos(x:extended):extended;
VAR
 e: Environment;
BEGIN
 ProcEntry(e);
 ArcCos:=arctan(sqrt(1.0-x)/(1.0+x));
 SetException(DivByZero,false);
 ProcExit(e);
END;
```

ProcEntry(e) saves the caller's environment in e and sets IEEE defaults, so exceptions cannot halt the routine. If $x = -1$, the computation of the right side of the assignment to ArcCos will signal DivByZero, even though ArcCos will be assigned the correct value, Pi/2. SetException(DivByZero, false) clears the DivByZero flag, so the caller never sees it. If $x > 1$ or $x < -1$, the computation of ArcCos will appropriately signal Invalid. The ProcExit procedure will resignal Invalid after restoring the caller's environment, so if the caller's environment calls for halts on invalid, the halt will occur.

Appendix F

ASCII Character Set, Reserved Words, Predefined Identifiers



This appendix contains the ASCII character set used by IP and lists of all reserved words and predefined identifiers used by IP.

ASCII Character Set

ASCII code	char	ASCII	char	ASCII	char	ASCII	char
0	□	32	space	64	@	96	`
1	□	33	!	65	A	97	a
2	□	34	"	66	B	98	b
3	□	35	#	67	C	99	c
4	□	36	$	68	D	100	d
5	□	37	%	69	E	101	e
6	□	38	&	70	F	102	f
7	□	39	'	71	G	103	g
8	□	40	(	72	H	104	h
9	□	41	)	73	I	105	i
10	□	42	*	74	J	106	j
11	□	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	□	46	.	78	N	110	n
15	□	47	/	79	O	111	o
16	□	48	0	80	P	112	p
17	□	49	1	81	Q	113	q
18	□	50	2	82	R	114	r
19	□	51	3	83	S	115	s
20	□	52	4	84	T	116	t
21	□	53	5	85	U	117	u
22	□	54	6	86	V	118	v
23	□	55	7	87	W	119	w
24	□	56	8	88	X	120	x
25	□	57	9	89	Y	121	y
26	□	58	:	90	Z	122	z
27	□	59	;	91	[	123	{
28	□	60	<	92	\	124	|
29	□	61	=	93	]	125	}
30	□	62	>	94	^	126	~
31	□	63	?	95	_	127	□

Instant Pascal Reserved Words

The following are Instant Pascal's reserved words. These appear in uppercase letters throughout the body of this manual.

AND	DOWNTO	GOTO	OF	REPEAT	USES
ARRAY	ELSE	IF	OR	SET	VAR
BEGIN	END	IN	OTHERWISE	STRING	WHILE
CASE	FILE	LABEL	PACKED	THEN	WITH
CONST	FOR	MOD	PROCEDURE	TO	
DIV	FORWARD	NIL	PROGRAM	TYPE	
DO	FUNCTION	NOT	RECORD	UNTIL	

The word FORWARD is defined by the ANSI Standard to be a **directive**, rather than a reserved word. The major difference is that a directive can be redefined by a program, whereas a reserved word can't. Even though it is possible to redefine a directive, it should never be necessary and is poor programming practice.

Instant Pascal Predefined Identifiers

These are the identifiers of the predefined procedures, functions, types, and variables of Instant Pascal. The list does not include those types and identifiers that are declared or defined in the SANE unit (with the exception of the real-types).

If you declare or define one of these identifiers in your program, no error will result but you will lose the capability of the built-in or predeclared object defined by that identifier.

Constants

false	*maxlongint*
INF	*pi*
maxint	*true*

Types

boolean	*double*	*longint*	*text*
char	*extended*	*real*	
comp	*integer*	*single*	

Variables

input	*randseed*
output	

Procedures

close	*page*	*reset*	*write*
get	*put*	*rewrite*	*writeln*
new	*read*	*seek*	
open	*readln*	*setprefix*	

Functions

abs	*filepos*	*ord*	*sqrt*
arctan	*getprefix*	*pos*	*succ*
chr	*include*	*pred*	*trunc*
cos	*length*	*random*	
eof	*ln*	*round*	
eoln	*odd*	*sin*	
exp	*omit*	*sqr*	

Graphics

aqua
black
brown
darkblue
darkgreen
drawchar
drawline
drawstring
frameoval
framerect
getpenstate
gray1
gray2
green
h
invertcircle
lightblue
line
lineto
magenta
medblue
move
moveto
notpatbic
notpatcopy
notpator
notpatxor
orange
paintcircle
paintoval
paintrect
patbic
patcopy
pator
pattern
patxor
penmode
pennormal
penpat
pensize
penstate
pink
point
purple
rect
setpenstate
str255
v
vhselect
white
writedraw
yellow

Glossary

activation: The execution of a block of code. Normally a block will have either no activations (if it is not currently being executed) or one activation (if it is being executed). A block that is recursive may have more than one activation at a time.

actual parameter: A variable within a program that is passed to a procedure for processing. Compare **formal parameter**.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task.

allocate: To reserve space in memory for the storage of variables.

arithmetic operator: A symbol used in mathematical calculations.

array: A collection of variables that are declared with a common name in a single variable declaration. The variables all have the same type, which is called the **component type** of the array. Each variable is identified by means of an **index**, which is given in square brackets after the name of the variable. The index indicates the position of the variable in the array.

array element: An individual member of an array.

ASCII: Acronym for *American Standard Code for Information Interchange*, pronounced *Ask' ee*, a code in which the numbers from 0 to 127 stand for text characters—including the letters of the alphabet, the digits 0 through 9, punctuation marks, special characters, and control characters—used for representing text inside a computer and for transmitting text between computers or between a computer and a peripheral device.

assembly language: A low-level programming language in which individual machine-language instructions are written in a symbolic form more easily understood by a human programmer than machine language itself.

assignment: The process of giving a value to a variable or function.

base: The number of digits in a number system. Also see **radix**.

base type: The potential values of a set variable.

binary operator: An operator that combines two operands to produce a result; for example, * is a binary arithmetic operator, < is a binary relational operator, and OR is a binary logical (or boolean) operator. Compare **unary operator**.

block: A module of a Pascal program that includes a definition part, a declaration part, and a statement part.

block structure: The way in which Pascal is designed to allow the programmer to create discrete modules of code that can be used independently within a large program.

boolean operator: An operator, such as AND, that combines logical values to produce a logical result. Also known as a **logical operator**.

boolean type: A Pascal data type whose members may have the values *true* or *false*.

call: To invoke a procedure or function.

cardinality: The number of distinct values contained within an ordinal type.

case constants: The labels used to select options within a CASE statement.

char type: The data type that is used to hold **ASCII** character values.

character: A letter, digit, punctuation mark, or other symbol used in printing, displaying, or transferring information.

comment: An explanatory note about an Instant Pascal program that is ignored by the interpreter. Comments are written to convey information about the content and purpose of a program to human readers.

compiler: A language translator that converts a program written in a high-level programming language into an equivalent program in some lower-level language (such as machine language) for later execution. Compare **interpreter**.

compound statement: A series of statements enclosed within a BEGIN and an END that form a unit and are treated as one block of text by the interpreter.

concatenate: Literally, "to chain together." To combine two or more strings into a single, longer string containing all the characters in the original, individual strings.

conditional statement: A statement that will execute only when certain specified conditions are met.

congruent type: A type that is identical to or a subset of another type.

constant: A symbol in a program that represents a fixed, unchanging value. Compare **variable**.

control variable: A variable whose value determines the number of times an iterative statement is executed.

controlling expression: An expression the value of which determines the number of times an iterative statement is executed.

declaration: The part of a Pascal program in which labels, constants, types, and variables are defined.

decrement: To decrease a value by a set amount. Opposite of **increment**.

default: A value, action, or setting that is assumed or set in the absence of explicit instructions otherwise.

delimiter: A character that is used for punctuation to mark the beginning or end of a sequence of characters, statements, or expressions, and which therefore is not considered part of the sequence itself.

denormalized number: A number represented in floating-point format in which the first bit of the signficand is a zero. Compare **normalized number.**

directive: A Pascal symbol that is similar to a reserved word but that may be redefined by the programmer.

domain: the type of a pointer variable.

dynamic allocation: The process of reserving storage space in memory while a program is executing. Compare **static allocation**.

dynamic variable: A variable used to allocate storage space in memory during execution of a program. See **pointer variable**, **dynamic allocation**.

embedded: Contained within.

enumerated type: A type that is created by a programmer. An enumeated type consists of an ordered collection of identifiers listed in a type definition.

error: A condition whose outcome is undefined.

execute: To perform the actions specified by a program command or sequence of commands.

exit condition: A circumstance that causes a program block to stop executing.

exponent: In scientific notation, the number that denotes the power of ten to which a number is raised.

expression: Any representation of a value. This can be a single identifier, such as a numeric constant, or a more complex form that includes operators and several operands.

external file: A structured variable used to send and receive information from an external device, such as a disk drive.

field: One discrete variable within a **record**.

file variable: A structured variable, generally used to store information (either temporarily, within memory, or permanently, on an outside storage media).

fixed-point: A method of representing numbers inside the computer in which the decimal point (more correctly, the binary point) is considered to occur at a fixed position within the number. Typically, the point is considered to lie at the right end of the number so that the number is interpreted as an integer. Compare **floating-point**.

floating-point: A method of representing numbers inside the computer in which the decimal point is permitted to "float" to different positions within the number. Some of the bits within the number itself are used to keep track of the point's position. Compare **fixed-point**.

flow of control: The order in which Pascal statements within a program are executed. A general reference to the type of statements that control this order.

formal parameter: In the declaration of a procedure, the parameter that will be used to pass information into the procedure for processing. Compare **actual parameter**.

forward definition: A declaration that allows a procedure to be called (as in **indirect recursion**) before it is formally defined.

free-union variant: A variant part of a record that is defined without the use of a tag field.

function: A Pascal subroutine that returns a single value.

global variable: A variable whose value is available for use throughout an entire program.

graphics: The screen representation of forms other than the standard character set.

host type: The type that contains a subrange type.

I/O: The abbreviation for "input/output." A general term for the transfer of information into and out of the central processor of the computer.

identifier: The name given to a **variable**, **constant**, **data type**, **label**, **procedure**, or **function** and declared in the declaration part of a block.

increment: To increase a value by a set amount. The opposite of **decrement**.

index value: The value used to access one element of an array.

indirect recursion: A situation in which a subprogram calls a second subprogram, which in turn calls the first subprogram. Synonymous with **mutual recursion**.

infinity: A special bit pattern produced by SANE when a mathematical calculation should result in an exact mathematical infinity or when a calculation produces a number with magnitude too great for the intended real-type format.

initial value: The beginning value of the control variable in a FOR loop.

initialize: To set to an initial state or value in preparation for some computation. For example, to set the value of a variable at the beginning of a program.

integer: The data type that is the set of all whole numbers between −32767 and 32767.

integer-types: The *integer* and *longint* data types.

interpreter: A language translator that reads a program instruction by instruction and immediately translates each instruction for the computer to carry out. Compare **compiler**.

iteration: One repetition of a statement or block.

label: An identifier that consists of an integer in the range 0..9999 that is used by the GOTO statement.

limit expression: The expression that controls the number of times a FOR statement is executed.

limit value: The value of the limit expression in a FOR statement.

local variable: A variable that is defined for use only within the block in which it is declared.

logical operator: An operator, such as AND, that combines logical values to produce a logical result. Also known as a boolean operator. Compare **arithmetic operator, relational operator**.

machine language: The form in which instructions to a computer are stored in memory for direct execution by the computer's processor. Each model of computer processor (such as the 6502 microprocessor used in the Apple II family of computers) has its own form of machine language.

member: One element of an array or set.

mutual recursion: A situation in which a subprogram calls a second subprogram, which in turn calls the first subprogram. Synonymous with **indirect recursion**.

nested: An element (such as a statement or a block) that is contained within a like-structured program element. For example, an IF statement that is contained within the compound statement that may follow the reserved word THEN.

non-structured variable: A variable belonging to a simple data type.

normalized number: A number represented in floating-point format in which the first bit of the signficand is a one. Compare **denormalized number.**

null: An undefined value. Not equivalent to zero.

operand: A value to which an **operator** is applied.

operator: A symbol (such as + or *mod*) that stands for an operation to be performed on values.

ordinality: The quality of an ordered, linear relationship.

overflow: The condition that exists when an attempt is made to put more data into a memory area than it can hold.

packed: A space in memory allocated to a variable from which all the unused area has been removed.

parameter: A special kind of variable used by a procedure or function.

parameter list: The variables declared in the heading of a procedure or function.

Pascal: A high-level programming language with statements that resemble English sentences. Pascal was designed to teach programming as a systematic approach to problem solving. Named after the philosopher and mathematician Blaise Pascal.

physical addresses: The binary or hexadecimal references to memory space.

pointer variable: A variable whose value consists of the memory address of some other item. In Pascal, a special data type that allows the programmer to allocate memory dynamically.

positional notation: The system that defines the value of a digit relative to the radix point.

precedence: The order in which operators are applied in evaluating an expression.

precision: The number of digits allowed to the right of the decimal place in a real number.

predecessor: In an ordinal type, the element of the type that comes before the current element.

predefined procedures and functions: Specialized programs that are included as part of a language.

procedure: A block of code that performs a specific task as part of a larger program.

procedure body: The executable statements within a procedure.

procedure definition: The heading, declaration, and statement parts of a procedure.

procedure heading: The first line of a procedure definition that contains the procedure's name and formal parameter list.

radix: The number of digits in a number system. See also **base**.

radix point: The symbol that separates the integer and fractional parts of a real number. In the base ten system, the symbol is called the decimal point; in the base two system, the binary point.

real number: A number that may include a fractional part; represented inside the computer in **floating-point** form. Compare **integer**.

real-type: One of the set of Instant Pascal data types that are used to represent real numbers.

record: A structured variable that contains discrete fields that can be of different types. Records may be operated on using the WITH statement. See also **variant record**.

recursion: See **recursion**.

reference parameter: See **variable parameter**.

relational operator: The operators used to form expressions that compare one operand to another. Compare **arithmetic operator**, **logical operator**.

repetition statements: The statements in Pascal that cause an action to be repeated until a condition is met.

reserved word: A word or sequence of characters reserved by a programming language for some special use, and therefore unavailable as a variable name in a program.

round: To change a real value to an integer value. The Pascal function that performs this operation.

round-off error: An error that can occur when a real number loses significant digits in the process of being converted from one number system to another.

SANE: Acronym for the Standard Apple Numeric Environment. SANE is the foundation for all arithmetic operations in Instant Pascal and for the enhanced real-types: *double*, *extended*, and *comp*.

scalar type: A data type whose members belong to an ordered range of values. Synonymous with ordinal type.

scope: The range in a program in which an **identifier** can be recognized.

semicolon: The delimiter used in Pascal to separate semantic entities.

set: A Pascal structured type that is a non-ordered group of less than 256 values, each of which must be of a single **base type**.

set constructor: A constant that appears in the list of set members that is contained between square brackets in a set definition.

significand: In floating-point notation, the number that is multiplied by the exponent to yield the final value of the number.

simple data types: Any of the standard data types that hold a single value. Compare **structured data types**.

source text: The words and characters typed into the IP Text Window.

statement: A unit of a program in Pascal that specifies an action for the computer to perform.

string: An item of information consisting of a sequence of text characters. A special data type in Instant Pascal used to store sequences of characters.

structured data types: The Pascal data types **array**, **set**, **record**, and **file**. The structured types provide alternate means of storage and access of values. Compare **simple data types**.

subexpression: A portion of an expression that can be enclosed in parentheses to ensure that it is evaluated independently of the main expression.

subrange type: A type that consists of a set of values that is a subset of the values of another **ordinal type**.

subscript: The variable reference notation that appears in square brackets after an array identifier and that refers to a specific element of the array.

successor: In an ordinal type, the element of the type that comes after the current element.

symbol: A character or set of characters that has special meaning to the IP interpreter.

syntax: The rules governing the structure of statements or instructions in a programming language.

syntax diagram: A representation of a statement or structure that specifies all the possible forms the structure can take.

tag identifier: In a **variant record**, the field that indicates which group of variant fields is being used at any one time.

termination: The completion of the execution of a block.

text: Information presented in the form of characters readable by humans. Also an Instant Pascal predefined file type.

truncate: To shorten by discarding a part; specifically, to convert a real number to the next closest-to-zero integer.

truth table: A diagram showing all possible values of an expression given a *boolean* operator and the input condition of each operand.

type: The definition of the range of values that are legal for a variable.

type clash: A mismatch of types in an expression, assignment, or subprogram call which results in an error condition.

type compatibility: The condition that results when two variables or expressions represent values of the same underlying type (or possible different subranges of the same underlying type.

unary operator: An operator that applies to a single operand; for example, the NOT operator. Compare **binary operator**.

value parameter: A parameter that is passed to a procedure by value, rather than by address.

variable: The symbol used in a program to represent a location in the computer's memory where a value can be stored. Compare **constant**.

variable parameter: A parameter that is passed to a procedure by address, rather than by value.

variant record: A record in which the number and type of fields may change during the course of program execution.

Bibliography

A number of excellent books have been written on the Pascal language. Here are a few that range from entry-level instruction to complete and thorough descriptions of the language.

Reference Books

These books are the sources that define "standard Pascal."

Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*, 3rd ed. Revised by: Andrew B. Mickel and James F. Miner. New York: Springer-Verlag, 1985.

This book is the original documentation for the language as provided by Niklaus Wirth. The first section of the book ("User Manual") presents some of the main theories behind structured programming. The second section ("Report") is the formal description of the language. The third edition has been revised to include information on the International Standards Organization (ISO) version of the language.

Henry Ledgard, ed. *The American Pascal Standard (with Annotations)*. New York: Springer-Verlag, 1984.

The American National Standards Institute brings together opposing factions in the computer world to decide on what defines a "standard" implementation of a language. Instant Pascal is an ANSI Pascal—that is, it conforms to the rules of the language as described in this book. The annotations (by Henry Ledgard) explain some of the more difficult points and offer enlightening examples.

Doug Cooper. *Standard Pascal User Reference Manual*. New York: W. W. Norton and Co., 1983.

A well-written guide to standard Pascal by one of the authors of *Oh! Pascal!* (See entry under next heading.)

General Pascal Text Books

Robert Moll and Rachel Folsom, *Instant Pascal: An Introduction to Programming*. Boston, Mass.: Houghton-Mifflin, 1985.

A comprehensive and easy-to-follow textbook that introduces good programming practice along with the Pascal language. Written specifically for use with Instant Pascal.

Michael Clancy and Doug Cooper. *Oh! Pascal!* New York: W. W. Norton and Company, 1982.

One of the most thorough and entertaining introductions to Pascal.

Peter Grogono. *Programming in Pascal*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1978.

Another comprehensive and well-written text book, used in many introductory Pascal college courses.

Programming Methodology

Henry F. Ledgard, et al. *Pascal With Style (Programming Proverbs)*. Rochelle Park, N. J.: Hayden Book Company, Inc., 1979.

An excellent set of rules to code by. Written around a real-world design problem, this book leads you gently into good programming style.

Kenneth L. Bowles. *Microcomputer Problem Solving Using Pascal*. New York: Springer-Verlag, 1977.

Kenneth Bowles was head of the project at the University of California at San Diego that created what is now known as UCSD Pascal. The first widely used implementation of Pascal for small computers, UCSD Pascal brings the Pascal language together with a complete programming environment for microcomputers. This book discusses how Pascal's unique structures can be used to solve common types of programming problems.

Brian Kernighan and P. J. Plauger. *Software Tools in Pascal*. Reading, Mass.: Addison-Wesley Publishing Company, 1981.

This book focuses on the idea of software tools: programs that take advantage of Pascal's block structure to provide generic code for a specific purpose. *Software Tools* emphasizes structured programming and top-down design, as well as the importance of testing and documenting your code.

Index

Cast of Characters

A

B

C

D

E

I, J

K

L

M

N

V

W

X, Y, Z

Tell Apple About the Apple® II Instant Pascal® Language Reference Manual

Your Name: _____

Your Organization (If a school, specify grade level):

_____

Address: _____

_____

City/State/Zip: _____

We at Apple Computer use comments and suggestions from people like you to improve existing products and develop new and better products. Now that you've used this product, we want to know your suggestions and thoughts about your experience. Please use this form to tell Apple what you think.

Your Dealer Can Help: Apple can't respond to your individual questions. If you have a question, would like help, or need service, please contact your Apple dealer.

Part 1: Tell Apple About You

1. How much prior experience have you had using computers?
   ☐ None ☐ Little ☐ Moderate ☐ Extensive
2. What computers have you used? _____
   _____
   _____
3. Where do you use your Apple computer? Check all that apply. ☐ Home ☐ Work ☐ School
   ☐ Other _____
4. How many people in your school/office/household use the Apple computer?
   Adults _____ Children _____

Part 2: Tell Apple About Your System

1. Which Apple computer are you using?
   ☐ Apple IIe ☐ Apple IIc
2. Are you using Instant Pascal with a mouse or with the keyboard?
   ☐ Mouse ☐ Keyboard
3. Are you using a monitor or a television?
   ☐ Monitor ☐ Television
   What kind is it?
4. What programming languages, other than Instant Pascal, have you used? _____
   _____
   _____
5. Please list any other products (programs, operating systems, or languages) you use.

Name of Software	Version Number
_____	_____
_____	_____

☐ Monochrome (or black-and-white TV) ☐ Color

☐ Other Brand Name _____

Part 3: Tell Apple About This Product

Please take a few minutes to tell Apple what you thought of the *Apple II Instant Pascal Language Reference Manual.* Feel free to attach additional sheets.

1. Where did you buy this manual?

   ☐ Authorized Dealer ☐ Bookstore

   ☐ Through the Mail

   ☐ Other _____

2. Did you have any trouble finding a copy of this manual?

   ☐ Yes ☐ No

3. How have you read this manual?

   ☐ The whole manual from cover to cover

   ☐ Whole chapters at a time

   ☐ Only specific sections as needed

   ☐ Other _____

4. What did you like best about the manual?

   _____

   _____

   _____

5. What did you like least about the manual?

   _____

   _____

   _____

6. Which topics, if any, were confusing or not explained thoroughly enough? _____

   _____

   _____

7. Please describe specific problems you encountered with the manual. (Page numbers would be helpful.)

   _____

   _____

   _____

   _____

8. Which aids do you use to find what you're looking for?

   ☐ Index ☐ List of Figures and Tables

   ☐ Table of Contents ☐ Other _____

9. Overall, how would you rate this manual?

   Low High

   1 2 3 4 5 6 7 8 9 10

10. What other books have you used to supplement the information in this manual? _____

    _____

    _____

    _____

Thank you for your time and effort.

Apple II Instant Pascal Language Reference Manual

> $22.95 FPT
USA

The Official Publication from Apple Computer, Inc.

Instant Pascal® is Apple Computer's revolutionary new implementation of one of the world's most popular programming languages. Designed especially for learning, Instant Pascal (for the Apple® II family of personal computers) provides unique and innovative features that make programming fast and efficient.

The *Instant Pascal Language Reference Manual*, written for parents, teachers, students, and anyone needing in-depth information about Instant Pascal, is the complete, official reference manual from Apple Computer. Beginning with a discussion of overall program structure, it goes on to give detailed attention to every major aspect of the Instant Pascal language. In addition, it includes the following:

- Key sample programs and examples.
- Appendices providing technical specifications, including a description of the Standard Apple Numeric Environment.
- Instant Pascal syntax diagrams, tables, and illustrations.
- Coverage of such important concepts as procedures and functions; arrays, sets, and strings; records; pointers and dynamic variables; and files and input/output procedures.

The *Apple II Instant Pascal Language Reference Manual* is the essential and comprehensive description of the language that every serious Instant Pascal programmer will need.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
408 996-1010
TLX 171-576

030-0531-A

Addison-Wesley Publishing Company, Inc.

ISBN 0-201-17740-4